Handbook of Geometric Programming Using Open Geometry GL

Georg Glaeser
Hans–Peter Schröcker

SPRINGER

Handbook of

Geometric Programming Using Open Geometry GL

Springer
New York
Berlin
Heidelberg
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Handbook of

Geometric Programming Using Open Geometry GL

Georg Glaeser Hans-Peter Schröcker

With 221 Illustrations

Includes a CD-ROM 

Springer

Georg Glaeser
Chair for Geometry
University of Applied Arts, Vienna
Oskar Kokoschka-Platz 2
A-1010 Wien
Austria
georg.glaeser@uni-ak.ac.at

Hans-Peter Schröcker
University of Applied Arts, Vienna
Oskar Kokoschka-Platz 2
A-1010 Wien
Austria
hans-peter.schroecker@uni-ak.ac.at

Library of Congress Cataloging-in-Publication Data
Glaeser, Georg.
Handbook of geometric programming using Open Geometry GL/Georg Glaeser, Hans-Peter Schröcker.
p. cm.
Includes bibliographical references and index.
ISBN 0-387-95272-1 (alk. paper)
1. Computer graphics. 2. OpenGL I. Schröcker, Hans-Peter. II. Title.
T385.G5765 2001
006.6—dc21 2001049265

Printed on acid-free paper.

Production managed by Frank McGuckin; manufacturing supervised by Jeffrey Taub.
Camera-ready copy prepared from the authors' files.
Printed and bound by Maple Vail Book Manufacturing Group, York, PA.
Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-387-95272-1 SPIN 10832271

Springer-Verlag New York Berlin Heidelberg
A member of BertelsmannSpringer Science+Business Media GmbH

Preface

Overview

At the beginning of 1999, Springer-Verlag published the book Open Geometry OpenGL®+Advanced Geometry. There, the authors Georg Glaeser and Hellmuth Stachel presented a comprehensive library of geometric methods based on OpenGL routines. An accompanying CD-ROM provided the source code and many sample files. Many diverse topics are covered in this book. The theoretical background is carefully explained, and many examples are given.

Since the publication of Open Geometry, the source code has been improved and many additional features have been added to the program. Contributors from all over the world have come up up with new ideas, questions, and problems. This process has continued up to the present and Open Geometry is growing from day to day.

In order to make all of these improvements accessible to the public, and also in order to give deeper insight into Open Geometry, we decided to write this new Handbook on Open Geometry GL 2.0. It will fill certain gaps of Open Geometry 1.0 and explain new methods, techniques, and examples. On the accompanying CD-ROM the new source code and the sample files are included.

The Handbook now contains 101 well-documented examples and the reader is able to learn about Open Geometry by working through them. In addition, we present a compendium of all important Open Geometry classes and their methods.

However, we did not intend to write a new tutorial for Open Geometry. The Handbook is rather a sequel, written for the readers of the first book and for advanced programmers. Furthermore, it is a source of creative and good examples from diverse fields of geometry, computer graphics, and many other related fields like physics, mathematics, astronomy, biology, and geography.

Organization

In Chapter 1 we explain the philosophy and capacity of Open Geometry as well as the basic structure of an Open Geometry program. This is necessary to make the Handbook readable for advanced programmers who have not read the first book. Furthermore, it will be helpful to the reader in order to avoid programming style that might not be compatible with future versions.

Chapter 2 explains the most important 2D classes and provides new examples of animation and kinematics. We present enhanced methods of parameterized curves and conic section, and we introduce Béziers and B-spline curves in Open Geometry.

At the beginning of Chapter 3 we present a few basic 3D classes and explain in detail how to use the camera. In the following we include a wide section on diverse mathematical surfaces (our archives are full of them). Finally, the concept of spline surfaces will provide a powerful tool for practical applications.

The chapter on 3D Graphics is split because of our rich collection of new examples. One of the most important changes is the possibility to get high-quality views of Cad3D objects and to export them to other programs. We got some remarkable effects by using POV-Ray, a freeware ray-tracing program.

In Chapter 5 we show cross connections between Open Geometry and diverse geometric fields, such as descriptive geometry, projective geometry and differential geometry.

In Chapter 6 you can find a compendium of all important Open Geometry classes and all important methods. This has not been done till now, neither in the first book on Open Geometry 1.0 nor anywhere else. Dozens of new examples will explain the new methods and ideas that have been developed during the last year.

Guidelines for introducing new classes and good programming style are given in Chapter 7. Furthermore, we present examples of typical mistakes that weaken output quality and computing time. Usually these problems can be overcome by obeying only few simple rules.

The handbook would be useless without an accompanying CD-ROM. It includes the improved and enlarged source code and the complete listings of the new sample files. Thus, the reader will be able to visualize all the examples on her/his computer, and to explore the new possibilities herself/himself.

The book is intended for anybody interested in computer graphics. Like OPEN GEOMETRY 1.0, it can be used by students and teachers, both at high school or university, by scientists from diverse fields (mathematics, physics, computer graphics), and by people working in artistic fields (architects, designers). Of course, the reader must be able to write (or be willing to learn how to write) simple programs in C++.

Acknowledgments

Ever since the first book on OPEN GEOMETRY was released there has been considerable response from all over the world. Geometry courses based on OPEN GEOMETRY have been held in several countries. The ideas of many people are now integrated in Version 2.0.

We would like to thank the following individuals and institutions:

- Klaus LIST and Johannes WALLNER, Robert E. HARTLEY, Koichi HARADA and especially Jens DREGER for their intensive help with the LINUX version.
- Robert PUYOL for his Mac version of OPEN GEOMETRY and some important hints for the programming system.
- Darrel DUFFY (`http://master.neelum.com/~djduffy/`) for his idea to compile several scenes at one time.
- Michael BEESON (`http://www.mathcs.sjsu.edu/faculty/beeson/`) for the contribution of several highly interesting programs and many useful discussions.
- Hellmuth STACHEL, Peter PAUKOWITSCH, and their students for their contributions concerning the CAD3D system and CAD3D objects.
- Jörg PETERS for useful comments and ideas.
- Thomas GROHSER (Vienna, Austria) for his help in the early stages and at the very end of the production phase.
- Elisabeth HALMER for her help with the English version of this book.
- The Watcom Software Company in Waterloo (Ontario, Canada), especially Kathy KITSEMETRY. This company supported us with the Watcom C/C++ 11.0 compiler.
- Gerhard KARLHUBER (Vienna, Austria) for his text and code samples on elliptic compasses.
- The creators of OPENGL: The OPENGL team at SILICON GRAPHICS has been led by Kurt AKELEY, Bill GLAZIER, Kipp HICKMAN, Phil KARLTON, Mark SEGAL, Kevin P. SMITH, and Wei YEN.

- H. Pottmann and H. Wallner (again!) for contributing the ideas for Examples 4.13, 5.7, and 5.8. They stem from their book [27].
- Wendelin Degen for his valuable information on supercyclides.
- The POV-Ray Team (Steve Anger, Dieter Bayer, David K. Buck, Chris Carson, Aaron A. Collins, Chris Daily, Andreas Dilger, Steve Demlow, Alexander Enzmann, Dan Farmer, Timothy Wegner, Chris Young) for developing their wonderful freeware ray tracing system (`http://www.povray.org/`).
- Sebastian Michalski for his contribution of several sample programs.
- All those dozens of readers and students who gave us positive feedback via Email, supplied us with new ideas, and encouraged us to continue our work.

Contents

This page intentionally left blank

List of Figures

1

Introduction

This section is meant to introduce the gentle reader to the world of Open Geometry. We will talk about Open Geometry in general and about its history. We will explain the use of this handbook and how to start your first program.

The installation procedure of Open Geometry is described in Appendix D. However, there is no need to hurry. Just take your time and read carefully through the following sections. We will teach you everything you need to know to get started with Open Geometry. Later, in Section 1.4, we will come back to the installation process.

If you already have some experience with version 1.0 of Open Geometry, you may want to skip this introduction. Of course, we do not recommend this. But if you really cannot wait to run your first program with the new version you should at least have a quick look at Section 1.6. It will tell you about the most important changes and new features of Open Geometry 2.0.

1.1 What Is Open Geometry?

Open Geometry is a geometric programming system: With the help of a C++ compiler, the user is able to create images and animations with arbitrary geometric context. The system provides a large library that allows one to fulfill virtually any geometric task. The Open Geometry library itself is based on OpenGL, a system-independent graphics library that is supported by most compilers and

hardware systems.[1]

Additional to the rather basic but very effective OpenGL functions, the Open Geometry library provides the reader with 2D and 3D solutions to:

- the most common intersection and measuring problems,
- the creation of "sweeps" (e.g., path curves and path surfaces, i.e., sweeps of arbitrary curves),
- the creation of general solids by means of Boolean operations.

Open Geometry is a software environment that tries to link theory and practice of geometric programming. The user is able to realize direct geometrical thinking without having to care much about implementations. The idea is to write system-independent and readable graphics programs. Besides the code for the powerful geometric library, the attached CD contains C++ source code for more than 200 sample programs and executables for several platforms.

Summing up, Open Geometry:

- makes elementary and advanced geometric programming easy;
- fully supports the OpenGL standard (z-buffering, smooth shading, texture mapping, etc.),
- offers an advanced geometry library, with emphasis on kinematics and differential geometry;
- supports features like object generation by means of sweeping or Boolean operations.

Open Geometry was written for:

- students who want to get a deeper understanding of geometry;
- scientists who want to create excellent images for their publications;
- programmers who want to develop professional graphics software; and
- all people who love graphics and geometry.

[1] OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL supports a broad set of rendering, texture mapping, special effects, and other powerful visualization functions.

The History of Open Geometry

At the end of the eighties, Georg Glaeser and Hellmuth Stachel started with the idea of developing environments for geometers in order to allow them to develop their ideas more efficiently. Working closely together, thet took two different approaches: Hellmuth Stachel concentrated on writing CAD systems.[2] Georg Glaeser developed a programming environment for "geometry programmers," first in Pascal (SuperGraph, [11]), later on in C and C++ ([12]). Additionally, he wrote a book about fast algorithms for 3D Graphics in C ([13]).

At the end of 1998, Open Geometry 1.0 was first published ([14]), being a mixture of all the experiences made in these years.

In June 1999 Hans-Peter Schröcker began to work intensively with Open Geometry, and it soon turned out that with his help, the programming system improved amazingly. Thousands of lines of code flew from his "pen." This is why he is now coauthor of this book.

A main goal was to support as many systems as possible. The system in which Open Geometry was created initially is Windows NT (or Windows 2000 respectively). It is still *the* system, in which Open Geometry should run more or less bug-free. Windows 9x is also good, since it has the same user interface. The system was also successfully tested in a Linux environment. Updates can be found in the Internet. So please, have a look at the web page `http://www.uni-ak.ac.at/opengeom/` every once in a while.

1.2 Why Open Geometry?

There are quite a few 2D and 3D software applications on the market. Some of them are really professional graphics systems, providing excellent methods for the creation and manipulation of objects, both in 2D and 3D. Others specialize in the mathematical and geometrical background, and with their help, many scientific problems can be solved and visualized. In fact, the competition is stiff. So, why did we try to make another approach?

The thing is that Open Geometry is not intended to be "yet another graphics program." Open Geometry is a programming system that has another philosophy in mind: This philosophy is that the system is "open" and thus not restricted, since the user has direct access via programming. Menu systems can be wonderful, and they enable the user to do many things, but still, they are limited by nature.

Open Geometry has now been in development for a couple of years, and though its "body" has always been under control of a very few people, many programmers have contributed ideas and source code to it. These programmers come from

[2]The first result was the Cad3D system. The DOS version of this system comes with the CD.

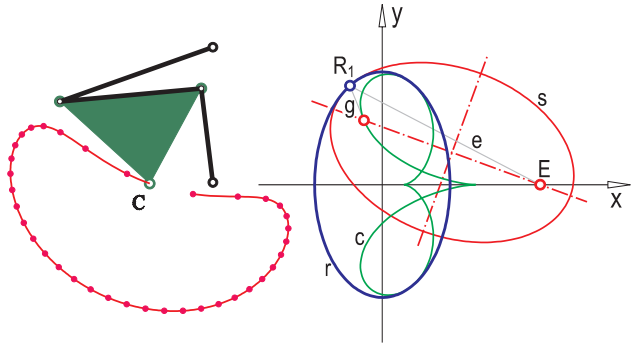

FIGURE 1.1. A four-bar linkage (left), a catacaustic (right).

various fields and from different educational backgrounds, but they always have had a strong affinity to geometry, and that is what makes OPEN GEOMETRY so appropriate for even the most complicated geometrical tasks.

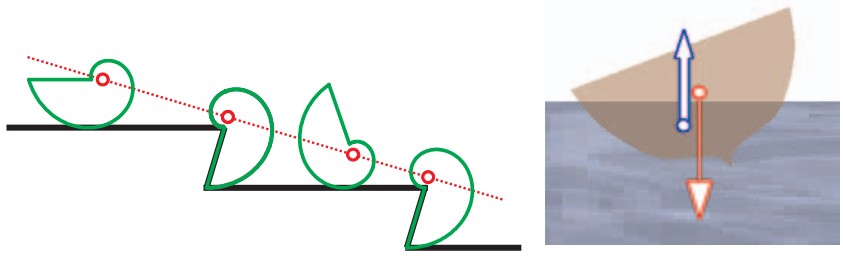

FIGURE 1.2. A rolling snail (logarithmic spiral) and a floating boat.

In the meantime, you can, e.g., easily:

- define a four-bar linkage and calculate path curves of arbitrary points connected with it (`"fourbar.cpp"`, Figure 1.1, left),
- display a caustic of a curved line both as the envelope of reflected light rays and as a path curve of the focal point of an osculating conic (`"catacaustic.cpp"`, Figure 1.1, right);
- illustrate how a logarithmic spiral rolls down some specially designed stairs (`"rolling_snail.cpp"`, Figure 1.2, left);
- simulate how a ship floats on a stormy ocean and, hopefully, does not sink (`"stormy_ocean.cpp"`, Figure 1.2, right);
- show how a tripod can be moved along a developable surface and, in a second stage, show conversly how the surface can be moved so that it always touches a fixed plane (`"oloid.cpp"`, Figure 1.3),

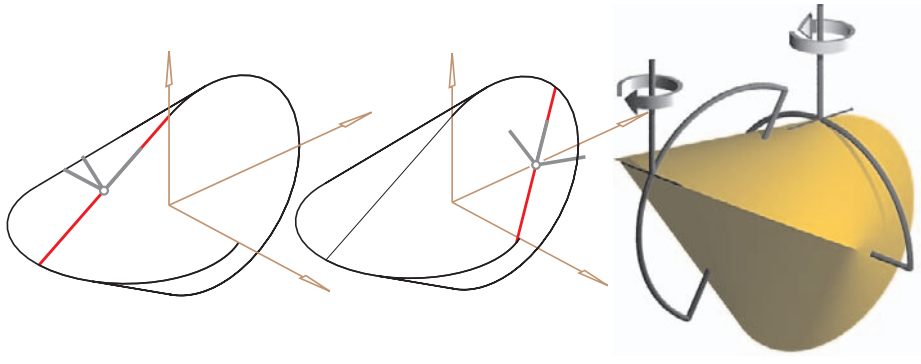

FIGURE 1.3. A moving tripod and the revolution of a "wobbler" (oloid).

- create a ruled surface as the path surface of a straight line and display its central line or even the normal surface along this line (`"normal_surfaces.cpp"`, Figure 1.4, left),
- let little arrows run around a MÖBIUS strip and thus show the surface's one-sided character (`"moebius_with_arrows.cpp"`, Figure 1.4, right),
- let all the above programs run in one menu-driven demo that can be executed in real time on virtually any modern computer (`"demo1.exe"`).

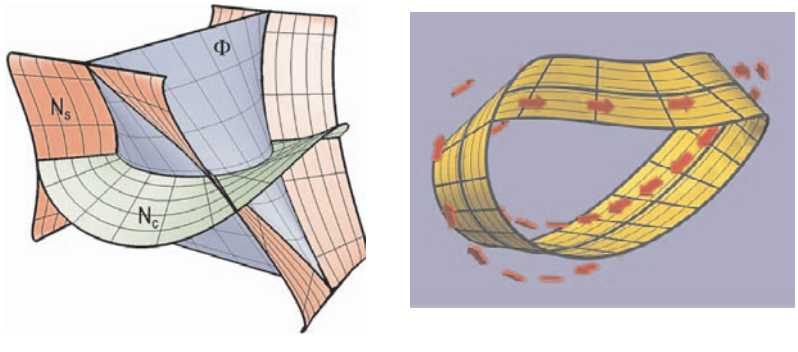

FIGURE 1.4. Ruled surface with normal surface (left), one-sided MÖBIUS strip (right).

1.3 About This Handbook and How to Use It

This handbook provides the reader with as much information about OPEN GEOMETRY (OPEN GEOMETRY 2.0) as possible. It is written for both beginners and advanced OPEN GEOMETRY users. One should be able to use the book without

having to read one chapter after the other. Nevertheless, it is recommended to skim over the introductory sections at the beginning.

The CD comes with quite a few executable demo programs. They are located in the `"DEMOS/"` directory and can be executed via mouse click. In general, they are multi scene applications with mostly about a dozen single applications compiled together. Before you work through a chapter, it is a good idea to browse through the corresponding demo programs. Then you have a good idea of what awaits you and which examples it might be worth focusing on.

An important part of the book is the index, which has automatically been produced by a self-written program.[3] In the index and throughout the book, OpenGL-keywords are written in *slanted letters*, Open Geometry-keywords are written in *italic letters*, and C-keywords are written in **bold letters**.

Our goal at the beginning was to describe every single class including all their member functions. This turned out to be almost impossible. Therefore, the listing in Chapter 6 is not complete. Still, we are convinced that we have described all those classes and functions that an Open Geometry user will ever work with. The residual classes were written only for internal use, and they are never used in application files. We also know that Open Geometry will continue developing, so that any update will again have undocumented classes.

Even though this book contains quite a lot of information, it no longer covers those theoretical topics the original one did (e.g., kinematics and Boolean operations). Thus, if you need more theoretical background in this respect, please refer to [14]. Of course, there are some entirely new topics that you can find in this book, e.g., sections about projective geometry, fractals or Bézier surfaces, and B-spline surfaces.

The present book contains lots of information that is not covered in [14]. This concerns mainly the dozens of new classes and functions that have been implemented since Version 1.0 was published. The sample files are quite different, which makes the book a useful addition to the first one. In the various subdirectories you can find dozens of new application files. You can view many of the corresponding executables with the enclosed demo-browser. We also created hundreds of new images, and the book has again become not only a handbook about software but also a "geometrical picture book."

All the applications of version 1.0 are compatible with the new version. Thus, you need not change anything in your programs written so far. Maybe minor corrections can be useful to support new features like the restart option or the multi scene option.

This book is full of source code listings. Usually we will indicate the source file at the top of the listing. The given path will be relative to Open Geometry's home

[3] The program "beauty.exe" is documented in [12].

directory (e.g., `"D:/OPENGEOM/"`) or to the subdirectory `"OPENGEOM/HANDBOOK/"`. The same is true for all other file names. That is, if you do not find a file in `"OPENGEOM/"`, you will find it in `"OPENGEOM/HANDBOOK/"`.

1.4 A First Program: Learning by Doing

Before we explain everything in detail, we want to get you into the kind of geometrical thinking that is well supported by OPEN GEOMETRY.

From school, you probably remember the following theorem:

Let A and B be fixed points, and C be an arbitrary third point. The three points have a well-defined circumcircle. When C runs on that circle, the angle γ in the triangle ABC will stay constant as long as C stays on the same side of AB. When C is on the other side, the angle is $180° - \gamma$.

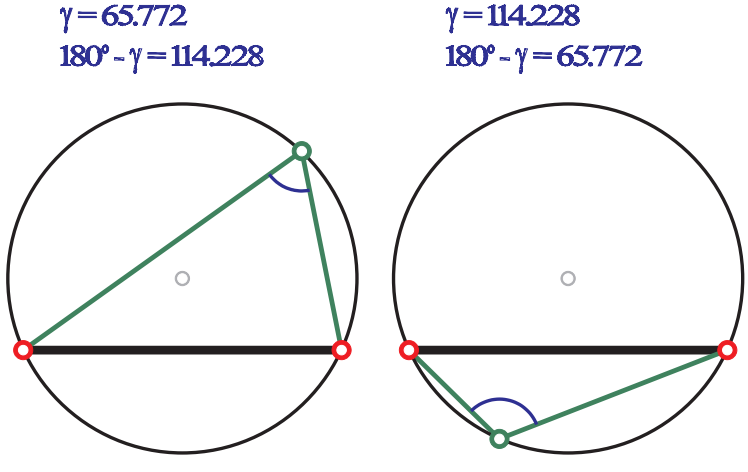

FIGURE 1.5. The angle at the periphery of a circle.

Now you want to write a little program that shows an animation of this (similar to Figure 1.5).

Geometrically speaking, you know what to do:

- Choose three points A, B, C;
- determine the circumcircle;
- calculate the angle $\gamma = \angle ACB$; and finally,
- move C along the circle (e.g., by means of a constant rotation about its center).

As a programmer, you know that it would be arduous to reinvent the wheel all the time. Therefore, use a programming system that provides all the basic tasks for you. You just have to know the commands, and — as you will see soon — these commands come quite naturally. Open Geometry is object-oriented. This means that the type of a variable is usually a class that not only contains data but is equipped with member functions (also called methods). Besides such classes, Open Geometry also supports "ordinary functions" that you probably know from OpenGL.

With this example we will try to get started with Open Geometry. We do it in several steps:

A. The simple program `"circumcircle.cpp"`

Take a look at the following listing, which you will understand immediately (we will not go into detail for the time being):

Listing of `"circumcircle.cpp"`:

```
#include "opengeom.h"
#include "defaults2d.h"

P2d A, B, C;
Circ2d CircumCircle;

void Scene::Init( )
{
   A( -4, 0 );
   B( 4, 0 );
   C( 3, 5 );
   CircumCircle.Def( Black, A, B, C );
}

void Scene::Draw( )
{
   StraightLine2d( Black, A, B, VERY_THICK );
   StraightLine2d( Green, A, C, MEDIUM );
   StraightLine2d( Green, B, C, MEDIUM );
   CircumCircle.Draw( THIN );
   A.Mark( Red, 0.2, 0.1 );
   B.Mark( Red, 0.2, 0.1 );
   C.Mark( Green, 0.2, 0.1 );
}
```

The program uses two predefined OPEN GEOMETRY classes, *P2d* ("two-dimensional points") and *Circ2d* (circle in 2D). They are declared via `"opengeom.h"`. There is another file `"defaults2d.h"` included that initializes a standard 2D window of "handy" size, which is about the size of the sheet of paper in front of you when you measure in centimeters, or about the size of your screen when you measure in inches. Three points and one circle are declared.

In an initializing part, we assign Cartesian coordinates to the points and calculate the circumcircle by means of the member function Def(...) of *Circ2d*. In the drawing part, we draw the sides of the triangle ABC and the circumcircle. *StraightLine2d*(...) is a predefined function; Draw(...) is a member function of *Circ2d*. Note the line styles and the different colors. Finally, we mark the points by means of the member function Mark(...). The output for the time being is to be seen in Figure 1.6.

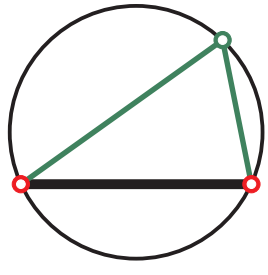

FIGURE 1.6. The output of `"circumcircle.cpp"`.

B. Compiling the program, simple changes in the program

At this stage you should install OPEN GEOMETRY on your computer (if you have not already done so). Appendix D will tell you how to do this. Furthermore, you will need to know how to compile and execute the above program.

Here is the recipe for the WINDOWS environment in connection with a `Microsoft Visual C++` compiler.[4] (If you work on another platform or with another compiler, you can still run the executable `"circumcircle.exe"` on your CD in order to understand the following.)

1. Load the workspace `mfcgl`.
2. Open the file `"try.cpp"`.
3. Be sure that except the first two **#include** statements all other lines are commented.

[4]The system is compatible with versions 5.0 and 6.0, where the compiler should not give any warning even at the highest warning level.

4. Search for the string `"circumcircle.cpp"`. Remove the comments on this line.
5. Build and execute the program. A graphics window should appear, similar to Figure 1.6.

Now, in order to get a feeling for the program, make a minor change:

- Add a fourth point M to the list of points.
- Initialize this point after the definition of the circumcircle:

  M = CircumCircle.GetCenter( );
- Add the line

  M.Mark(Gray, 0.15, 0.1);

  in the drawing part.
- Recompile and see what happens. (Clearly, the center of the circumcircle is additionally marked.)

Now you can test what can be done with the executable via menu or keyboard without any change: You can, e.g., move the image by means of the arrows on the keyboard, or you can export the image as an EPS or BMP file.

C. The animated program `"circumference.cpp"`

We now add a few additional commands and then animate the whole program. If you find it too bothersome to type everything, just load the file `"circumference.cpp"` almost exactly as in **B**, beginning with Step 3. The listing of the file is to be found below.

In order to show the angle γ, we use the function MarkAngle(...), which — as a side effect — returns the value of γ (in degrees).[5] Since the sign of the angle is of no relevance, we take the absolute value of γ (fabs(...)). For the information of the viewer, the angle is written on the screen with the function PrintString(...), which obviously works in the same way as the well-known C function printf(...). The Greek letter γ is displayed with the string $gamma$.[6] Additionally, the supplementary angle $180° - \gamma$ is displayed. In order to allow such a sophisticated output, we need to call the function HiQuality( ) in the Init( ) part. There, we also find the command AllowRestart( ), which — as the name tells — allows a restart of the animation at any time.

[5] In many cases Open Geometry prefers to work with degrees instead of arc lengths. Especially rotation angles are always given in degrees for better understanding.

[6] On page 562 you will find a more detailed description of how to print special symbols on the Open Geometry screen.

Listing of `"circumference.cpp"`:

```
#include "opengeom.h"

// Four points and a circle are global
P2d A, B, C, M;
Circ2d CircumCircle;

void Scene::Init( )
{
   A( -4, 0 ); // points of a triangle
   B( 4, 0 );
   C( 3, 5 );
   CircumCircle.Def( Black, A, B, C );
   M = CircumCircle.GetCenter( ); // center of circle
   HiQuality( ); // output allows Greek letters
   AllowRestart( ); // restart via menu is possible
}

void Scene::Draw( )
{
   StraightLine2d( Black, A, B, VERY_THICK );
   StraightLine2d( Green, A, C, MEDIUM );
   StraightLine2d( Green, B, C, MEDIUM );
   CircumCircle.Draw( THIN );
   Real gamma;
   // show the angle gamma as an arc
   gamma = fabs( MarkAngle( Blue, A, C, B, 1, 20, THIN ) );
   // show the value of the angle on the screen
   PrintString( Blue, -3, 8,
      "$gamma$=%.3f", gamma );
   PrintString( Blue, -3, 7,
      "180 -$gamma$=%.3f", 180 - gamma );
   // mark the points
   A.Mark( Red, 0.2, 0.1 );
   B.Mark( Red, 0.2, 0.1 );
   C.Mark( Green, 0.2, 0.1 );
   M.Mark( Gray, 0.15, 0.1 );
}

void Scene::Animate( )
{
   C.Rotate( M, 1 ); // C runs on the circle
}

void Scene::CleanUp( )
{
}
```

```
void Projection::Def( )
{
  if ( VeryFirstTime( ) ) // Initialization of the drawing window
  {
     xyCoordinates( −12.0, 12.0, −8.0, 8.0 );
  }
}
```

Do not forget to delete the line that includes `"defaults2d.h"`! At this stage, of course, you do not understand everything, and actually you do not have to. The strategy is "learning by doing". In general, it is probably a good idea to take a look at several demo files and compare what is written there with the output.

Since we are now doing something non-trivial (we animate the scene), we had to copy the contents of `"defaults2d.h"` into our program and modify them. Do not worry about this right now. We will explain everything later on. Just see what is done: The point C is rotated about a point M which is the center of the circumcircle. The rotation angle is small (only 1°), since the rotation is done for every new frame. Modern computers are fast enough to run this animation in real time, i.e., to display at least 20 images per second. The basic graphics output is done by OpenGL-routines that are invisibly used by the Open Geometry-library.

D. The third dimension – quite easy

Well, you might think, 2D problems are much easier to solve than 3D problems. Of course, you are right, but it would not be Open Geometry if it did not solve the same tasks in 3D – and that more or less by simply replacing all occurrences of 2d by 3d. Naturally, points in 3D should be initialized with three coordinates, and the point C must now rotate about the circle's axis. Finally, a 3D projection instead of a 2D window has to be initialized. But that is all in order to create a new Open Geometry-application `"circumference3d.cpp"`!

E.g., the function *StraightLine3d*(...) clearly draws a straight line in 3D, and the function MarkAngle(...) draws an arc in 3D when the parameters are space points *P3d*. The member function of the class *Scene*, DefaultOrthoProj( ), defines an ortho-projection in given direction.

Take a look at the listing of `"circumference3d.cpp"`. It illustrates that the theorem of the constant angle at circumference is also true, when C moves on an arbitrary circle in space that contains A and B.

Listing of "circumference3d.cpp":

```
#include "opengeom.h"

P3d A, B, C, M;
Circ3d CircumCircle;

void Scene::Init( )
{
   A( −4, 3, −3 ); // points of a triangle
   B( 4, 0, 3 );
   C( 3, 0, 5 );
   CircumCircle.Def( Black, A, B, C, 100 );
   M = CircumCircle.GetCenter( ); // center of circle
   HiQuality( ); // output allows Greek letters
   AllowRestart( ); // restart via menu is possible
}

void Scene::Draw( )
{
   StraightLine3d( Black, A, B, VERY_THICK );
   StraightLine3d( Green, A, C, MEDIUM );
   StraightLine3d( Green, B, C, MEDIUM );
   CircumCircle.Draw( THIN );
   Real gamma;
   // show the angle gamma as an arc
   gamma = fabs( MarkAngle( Blue, A, C, B, 1, 20, THIN ) );
   // show the value of the angle on the screen
   PrintString( Blue, −3, 8,
      "$gamma$=%.3f", gamma );
   PrintString( Blue, −3, 7,
      "180 -$gamma$=%.3f", 180 − gamma );
   // mark the points
   A.Mark( Red, 0.2, 0.1 );
   B.Mark( Red, 0.2, 0.1 );
   C.Mark( Green, 0.2, 0.1 );
   M.Mark( Gray, 0.15, 0.1 );
}

void Scene::Animate( )
{
   C.Rotate( CircumCircle.GetAxis( ), 1 ); // C runs on the circle
}

void Scene::CleanUp( )
{
}
```

```
void Projection::Def( )
{
  if ( VeryFirstTime( ) ) // Initialization of the drawing window
  {
     DefaultOrthoProj( 28, 18, 12 );
  }
}
```

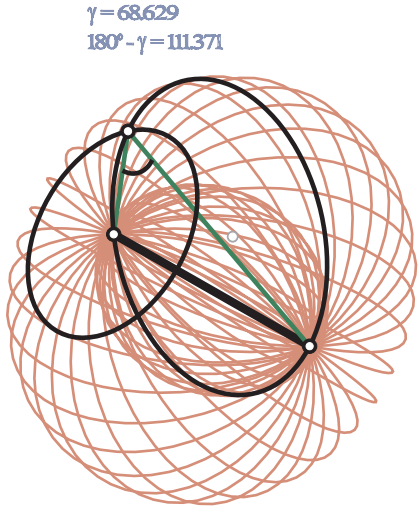

FIGURE 1.7. Torus as locus of space points with constant angle at circumference.

With only slight modifications, one can illustrate the following theorem by means of an animation (Figure 1.7):
The locus of all space points from which a line segment AB *is seen under constant angle at circumference* γ *is a torus.*
Have a look at the corresponding file `"torus_as_locus1.cpp"`. Since only few lines differ from `"circumference3d.cpp"`, you will soon understand the code. Here is the drawing of the meridians of the torus:

Listing from `"torus_as_locus1.cpp"`:

```
StrL3d rot_axis;
rot_axis.Def( A, B );
Circ3d meridian_circle;
meridian_circle.Def( Red, A, C, B, 50 );
int i, n = 30;
for ( i = 0; i < n; i++ )
{
```

```
    meridian_circle.Draw( THIN );
    meridian_circle.Rotate( rot_axis, 360.0 / n );
}
Circ3d parallel_circle;
parallel_circle.Def( Black, C, rot_axis, 70 );
parallel_circle.Draw( MEDIUM );
```

Obviously, a new class *StrL3d* – a straight line in 3D as a geometric object with member functions like Def(...) – is used. The 3D circle *Circ3d* can now be rotated about the axis AB. The axis can also be used to generate a parallel circle on the torus.

The animation part is adapted slightly so that the point C now runs somehow on the torus:

Listing from `"torus_as_locus1.cpp"`:

```
C.Rotate( CircumCircle.GetAxis( ), 2 );
StrL3d axis;
axis.Def( A, B );
C.Rotate( axis, 2 ); // C now runs on a torus
```

Try out what happens if you change the rotation angles or, e.g., the value of the variable n in the rotation of the meridian circle.

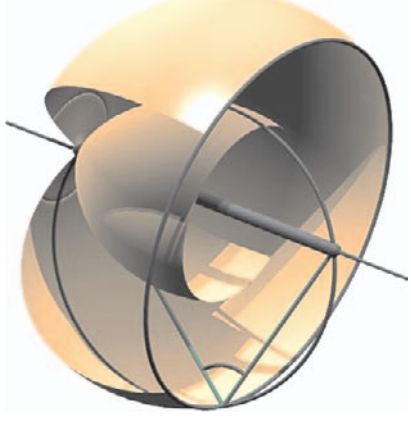

FIGURE 1.8. The illustration of the theorem about the angle at circumference in space (output of `"torus_as_locus2.cpp"`).

Finally, we want to display the described torus in a more sophisticated way (`"torus_as_locus2.cpp"`, Figure 1.8). The surface shall be displayed shaded. We, therefore, use an appropriate class and declare an instance of that class:

```
SurfOfRevol Torus;
```

In the initializing part, the surface is defined

Listing from `"torus_as_locus2.cpp"`:

```
Circ3d meridian_circle;
meridian_circle.Def( Red, A, C, B, 60 );
CubicSpline3d m;
meridian_circle.Copy( m, 15, 52 );
Torus.Def( Yellow, false, 60, m, -140, 125, StrL3d( A, B ) );
```

Of course, those few lines need some explanation – which will be given when surfaces of revolution are described in detail[7]. (You are free to experiment what happens if you change the parameters). The drawing of the surface, however, is done extremely simple by means of the single line. Additionally, the axis of the torus is displayed:

Listing from `"torus_as_locus2.cpp"`:

```
Torus.Shade( SMOOTH, REFLECTING );
Torus.GetAxis( ).Draw( Black, -3, A.Distance( B ) + 3, MEDIUM );
```

The torus now appears shaded on the screen[8]. The scene can automatically be manipulated via menu or keyboard. E.g., the light source can be rotated, one can display special projections like the top view, front view or right side view, zoom in or zoom out, etc.

You might have an ultimate question: Where do the cast shadows in Figure 1.8 come from? The answer is: The scene was exported to POV-Ray and rendered by this program. More about this new export utility of Open Geometry 2.0 can be found on page 335 in Section 4.2. There, you will find a short introduction to POV-Ray's scene description language and how you can use it to render Open Geometry scenes.

[7] In fact, it is not necessary to declare the torus as a surface of revolution. There exists an Open Geometry class called *Torus* that can do the same. Not all examples of this book use the simplest solution of a certain problem – be it for reasons of didactics or because the more advanced class has not been available when the program was written. We do not consider this as a drawback. On the contrary, you will get to know even more of Open Geometry's classes that way.

[8] "Shaded" usually means: No cast shadows.

1.5 The Structure of an Open Geometry 2.0 Program

After your first experience with OPEN GEOMETRY in the previous section, we are going to explain the basic structure of an OPEN GEOMETRY program. We encourage the beginner to follow our little step by step introduction live on a computer screen. Experienced OPEN GEOMETRY programmers may skip this section but perhaps even they will find one or the other useful hint.

We begin with a listing of a minimal OPEN GEOMETRY program for displaying a 2D scenery (`"minimal2d.cpp"`). You find it in `"USER/TEMPLETS/"`[9].

Listing from `"USER/TEMPLETS/minimal2d.cpp"`:

```
#include "opengeom.h"

void Scene::Init( )
{
}
void Scene::Draw( )
{
}
void Scene::Animate( )
{
}
void Scene::CleanUp( )
{
}
void Projection::Def( )
{
   if ( VeryFirstTime( ) )
   {
      xyCoordinates( -12.0, 12.0, -10.0, 10.0 );
   }
}
```

[9]Throughout this book, we will give path names relative to the standard OPEN GEOMETRY directory. I.e., the absolute path of the mentioned directory is something like `"D:/OPENGEOM/USER/TEMPLETS/"`

We suggest that you save this file under a different name (let's say `"simple.cpp"` in the folder `"USER/"`. In order to make it OPEN GEOMETRY's active file, you have to include it in `"try.cpp"` by inserting the line

#include `"USER/simple.cpp"`

(compare page 9).

Now you can compile and run `"simple.cpp"`. You will see that the code does nothing but open a window with a background in pure white, a few buttons and menu items. However, there are already a few things to mention:

- You have to include `"opengeom.h"` at the top of every OPEN GEOMETRY file. This is not an absolute must. But you won't be able to use any of OPEN GEOMETRY's classes if you don't.
- In order to compile the program without errors, you must call the functions Init( ), Draw( ), Animate( ), CleanUp( ) and Projection::Def( ).
- You have to initialize a drawing window in Projection::Def( ).

If you begin your new program with a templet file from the `"USER/TEMPLETS/"` directory, everything will be prepared and you need not bother about those things. Still, it is important that you know a few basic facts about each part of the above listing.

Usually it is enough to include `"opengeom.h"`. All other necessary files will be linked automatically to the OPEN GEOMETRY project. For some tasks, however, it is necessary to explicitly include an additional header file (e.g., `"bezier.h"`, `"fractal.h"`, `"kinemat.h"` or `"ruled_surface.h"`). Of course, you are free to write and include your own header files as well.

In Projection::Def( ), you define how the scene is displayed. You can choose between 2D and 3D views. In the above example you will see a 2D window that is adjusted in a way that the rectangle with vertices $(\pm 12, \pm 10)$ fits nicely on the screen. Depending on the shape of your OPEN GEOMETRY window, you will usually see a little more than that. You can adjust the default window size by pressing <`Ctrl+W`> in an active OPEN GEOMETRY window. This opens a dialogue box where you can set all window parameters.

If you want to see a 3D image, you have to replace Projection::Def( ) by

```
void Projection::Def( )
{
   if ( VeryFirstTime( ) )
   {
      DefaultCamera( 28, 18, 12 );
   }
}
```

This yields a standard 3D window with eye point at $(28, 18, 12)$ The target point is by default set to $(0, 0, 0)$. We suggest that you change your Projection::Def( ) to 3D right now. If you compile and run, you will see only small differences. The scene is still empty but some buttons and menu items are no longer disabled (Figure 1.9). They concern typical operations with eye point and light source that only make sense in three dimensions. You can explore them a little later when we have added objects to our scene.

FIGURE 1.9. The Open Geometry button bar.

Note that (theoretically) the difference between 2D and 3D applications can only be seen in Projection::Def( ). In all other parts you can use both, 2D and 3D objects. Occasionally, this can be quite useful (compare Example 2.24).

Now it is time to add an object to our scene. You can, e.g., visualize Open Geometry's coordinate system by inserting

```
ShowAxes3d( Black, 10, 11, 12 );
```

in the Draw( ) part. This displays the coordinate axes in black color. At their end points $(10, 0, 0)$, $(0, 11, 0)$ and $(0, 0, 12)$ little arrows and the letters "x", "y" and "z" are attached. Now you can try some of the buttons in the Open Geometry window. Especially those for moving the camera will be of interest.

Adding of other geometric objects is usually very simple. In order to add, e.g., a box centered at the origin of the coordinate system to our scene, we just add the following lines to Draw( ):

```
Box cube;
cube.Def( Red, 8, 8, 8 );
cube.Translate( -4, -4, -4 );
cube.Shade( );
```

We tell the compiler that cube is an instance of the object *Box*. In the next line we define its color and its dimension in x-, y- and z-direction. Finally, we translate it by the vector $(-4, -4, -4)$ in order to center it around the origin and display it on the screen.

So far, so good – there is just one thing: Later, we will make a lot of animations, i.e., the scene will undergo certain changes. We will have to draw a first frame, a second frame, a third frame and so on. For every new frame the relevant changes of the scene must be recomputed.

For a real-time animation, you will need at least 20 frames per second. This is no problem with sceneries of little or moderate complexity (as in our example). If, however, many complicated objects (e.g., a smoothly shaded parameterized surface with hundreds of triangles and a contour outline) occur, you may soon reach the limits of your hardware.

An OPEN GEOMETRY program starts by calling Init( ) and Draw( ). If you see the first frame you can start an animation by either pressing <Ctrl+F> or by clicking on the button labelled Fps (frames per second). This causes an alternate calling of Draw( ) and Animate( ) until the user stops the animation or closes the OPEN GEOMETRY window. Hence, the commands of these two functions have to be executed every time before a new frame can be displayed. If there are some tasks that need not be repeatedly performed you can (and should!) place them somewhere else. It is, e.g., not necessary to define and adjust the box in Draw( ). A good alternative code would look as follows

```
#include "opengeom.h"

Box Cube;

void Scene::Init( )
{
   Cube.Def( Red, 8, 8, 8 );
   Cube.Translate( -4, -4, -4 );
}
void Scene::Draw( )
{
   ShowAxes3d( Black, 10, 11, 12 );
   Cube.Shade( );
}
void Scene::Animate( )
{
}
void Scene::CleanUp( )
{
}
void Projection::Def( )
{
   if ( VeryFirstTime( ) )
   {
      DefaultCamera( 28, 18, 12 );
   }
}
```

We define the cube globally before Init( )[10]. This means that all following functions such as Init( ), Draw( ), and Animate( )will have access to Cube. In Init( ) we define and translate the cube while only the shading is left to Draw( ). Note that the definition could also be done by means of a constructor, i.e., together with the declaration of Cube. However, we do not recommend this in general (it does not really increase the code's readability, and, what is more important, might impede the restarting of the program). Note further that the drawing and shading has to be done in Draw( ). If you do it somewhere else, you will not get an error message but you won't be able to see the object you want to display[11].

Finally, we write a small animation and insert

```
Cube.Rotate( Yaxis, 3 );
```

into Animate( ). With every new frame the cube will be rotated through 3° about the y-axis of the coordinate system. You can switch to the next frame by pressing <Ctrl+N> in the Open Geometry window. Alternatively, you can press <Ctrl+F> or <Ctrl+R> to start the auto–animation or the auto–animation plus rotation about the z-axis. There exist menu items and buttons for these commands as well. You may try to identify them – a wrong guess can't do any harm.

Note the fundamental difference between inserting a line like

```
Cube.Rotate( Xaxis, 45 );
```

in Init( ) and in Animate( ). In the first case, the cube will be rotated only once about x while, in the second case, it will be rotated with every new frame. The lines in Animate( ) may as well be written in Draw( ) and sometimes we will do that (because the part is very short or because we want the start with scene that has already changed). In general, it is better for a clear and understandable code to separate drawing and animation.

The only remaining part of an Open Geometry program is CleanUp( ). In fact, this part is not used very frequently. It is called only once at the very end of the program. Perhaps you need global access to dynamically allocated memory. Then you can, e.g., free the memory in CleanUp( ). However, this is an advanced topic and shall be skipped at this place.

Finally, if you only want to do some drawing (no animation, no memory allocation, no special camera position), you can use the templets `"USER/TEMPLETS/minimal2d.cpp"` or `"USER/TEMPLETS/minimal3d.cpp"`.

[10] As a general rule, we will write the first letter of a global variable in uppercase format. Therefore we changed the name from cube to Cube.

[11] Well, if you want to obscure your ideas, you can do it by calling drawing routines in Animate( ).

There, a file (`"defaults2d.h"` or `"defaults3d.h"`, respectively) that takes responsibility for all obligatory function calls except Init( ) and Draw( ) is included. You yourself need not worry about anything.

1.6 What Has Been Changed in the New Version?

Ever since OPEN GEOMETRY has been released, hundreds of minor changes or corrections and dozens of implementations of new classes have been made. You will not be told about the minor corrections, but you will find the description of all the new classes in the "compendium". The main goal has always been to stay compatible with the first release.

- A change that has to be mentioned here is the following: In the function Projection::Def( ) we used to have question

  if (FrameNum( ) == 1) ...

  This has been changed to

  if (VeryFirstTime( )) ...

  for the following reason: When a scene is started and the first frame is drawn, one can interactively make some non-relevant changes (like rotating the camera or changing the light source or simply resizing the drawing window) that actually have nothing to do with the definition of the geometrical objects. In such a case, when the scene is redrawn, FrameNum( ) will still be 1. So any change of the camera, e.g., would be undone in Projection::Def( ), which can be nasty.[12]

- Some minor changes that shall be mentioned concern the menu system. The menu item "Program" now has two more sub-items "Program→Restart program" and "Program→Show source code", the menu item "Help" a new sub-item "Help→Display initializing file 'og.ini'"

  In the status bar you can now see whether you run a 2D application or a 3D application. In the 2D case it is just the hint 2D, in the case of a 3D application, however, you will see the word Perspective when the eye point is not infinite or the word Ortho3d when a normal projection is applied.

- Besides those small changes, there are several improvements that count much more: One of them is that there is now a way to restart a program at any time, if some rules are obeyed (those rules are explained in Section 7.2). When you add the line

  AllowRestart( );

[12] Don't worry, we are still compatible with the old style. But we would be glad, if you erase it from your templet files and do not use it any longer.

in the Init( )-part, you can restart the program either via keyboard (<Crtl+Shift+r>) or the menu item "Program→Restart program".[13]

- Another improvement is that you can now export 3D scenes as EPS-files. If the result was always correct, this would be a real "revolution", since this would mean that we can generate 3D images with extremely high resolution. However, there are some restrictions that decrease the value of this new feature: The drawing order must be back to front in order to create correct images. Still, simple 3D scenes can now be stored in the EPS format and then been manipulated with other professional programs (or a text editor, if you are a Post Script hacker).

- A change that is really notable, is the possibility of exporting scenes to POV-Ray. This opens the large world of ray-tracing to scenes that were generated via Open Geometry. Many of the images in this book have been rendered by POV-Ray.

- One of the major improvements in Open Geometry 2.0 is the possibility of compiling several programs at once (compare Section 7.1). Having done this, you can leaf through the programs via menu item, or shortcut key.

  Imagine, e.g., that you are preparing a public presentation. You write your sample files, compile them at once and show them to the audience without any need to compile during your speech. Or perhaps you want to display Open Geometry scenes on a remote computer with no Open Geometry installed. You can compile your scenes, create a *single* `"*.exe"` file and run it wherever you like.

1.7 What to Keep in Mind from the Beginning

How to use Open Geometry

Every Open Geometry application is written in C++. This has the advantage that your application is as good as you are able to tell the computer your geometrical ideas via programming commands. In practice, this means: The more Open Geometry commands and Open Geometry classes you know, the quicker you will get the best results.

It also means, however, that you have to know some basics about C++. In fact, it is not much you have to know – you will learn more and more by just taking a look at simple demo files.

[13] You can automatize this with a single line in the file `"og.ini"`. More about this in 7.3.

In order to explore OPEN GEOMETRY's classes, you can have a look at the header-files in the `"H/"`-directory. Usually, you should be able to guess the contents from the filenames. Alternatively, you can search this directory for a string like

`"class Sphere"`.

This will lead you directly to `"sphere.h"` where you will find *all* constructors, operators, methods and member variables of the class *Sphere*, whether they are **private**, **protected** or **public**. Most of them are listed in Chapter 6 of this book. The remaining undocumented classes should hardly be of relevance to you.

If you want to have a look at the implementation of a method, search, e.g., for the string

`"Sphere::GetContour"`.

You will find it together with the implementation of the corresponding method in `"C/o.cpp"`.

The advanced programmer may get a deeper understanding of OPEN GEOMETRY by studying the source code. But, please, do not change it! If you want to use classes of your own, just read on. Very soon, you will be told how to integrate them in OPEN GEOMETRY.

Programming style

You can, of course, use any programming style that C++ supports. You can break lines almost wherever you want, use any indent you want, keep space between braces or brackets or not, etc., etc.

We also do not tell you whether you should write every command into a single line or not, or how many comments you should add. Nor do we force anyone to use readable names – though this helps a lot when other persons skim over the code. Of course, you need not use an initial capital letter for your global variables as we did throughout the book.

The main thing that counts is to write solid code. We only ask you to keep a few things in mind before you seriously start to program, in order to

- stay compatible to future versions of OPEN GEOMETRY,
- write applications that can later on be adapted for a multi scene application.

Please read this:

Here are some rules that should be obeyed. Otherwise, you might not stay compatible with updates:

- Please, do not change any of the files you can find in the `"C/"` directory. These files with the meaningless names `"d.cpp"`, `"e.cpp"`, `"f.cpp"`, etc., are only there in order to enable you to compile the system with your personal C++ compiler. If you find any bug in one of these files, please report it to us, and we will supply you with the updated code. The same is true, if you have a good idea of how to expand a given class. In order to add your personal implementations, please use the (almost empty) file `"add_code.cpp"` in the directory `"USER/"`. We will give an example later on.

- Something similar is true for the header files in the `"H/"`-directory. You are asked to put new header files into the subdirectory `"H/"` of the `"USER/"` directory. The reason for this is that we unconsciously might have the idea of giving a future header file the same name as you did, and an update would overwrite your file.

- Use the variable type *Real* instead of **double**. For the time being, this is no must, at all, since both types are identical in the current version. This may change, however, when future hardware supports 64-bit floating point variables.

This page intentionally left blank

2

2D Graphics

Now that you learned about the basics of Open Geometry programming, you can work trough many new examples and sample files. In order to give a certain structure to this book, we start with examples from geometry in two dimensions. Many of Open Geometry's 2D classes and methods will be presented in this chapter. You will see:

- Open Geometry's basic 2D classes (points, straight and curved lines, circles, conics, polygons,...);
- simple and complex computer animations of physical concepts and kinematical devices;
- fractal programming with Open Geometry;
- free-form curves (Bézier curves and B-splines);

and much more.

Note that there is no real need to study all the examples, one after the other. You can skip any of them; cross references and the index always allow you to find additional information in other parts of this book. If you have a look at the examples, you will probably agree that many of them are quite interesting and not trivial at all. We are confident that we shall convince you that Open Geometry is really a versatile geometric programming system.

Additionally, you can find demo executables in the `"DEMOS/"` directory that allow one to have a look at the output of the sample programs without compiling them all.

2.1 Basic 2D Classes

To begin with, we will give a short introduction to Open Geometry's basic 2D classes by describing a few rather simple examples. However, we do not give a full description of the respective class. This is left to Chapter 6, where the reader will find a header listing and a detailed description of all the important classes and functions. Usually, we will refer to a certain Open Geometry sample program. Therefore, it is advisable to have your computer turned on while reading the following pages.

Our starting example already presents many basic classes like points, straight lines, circles, and conic sections:

Example 2.1. Three planets
In `"three_planets.cpp"` we display the path curves of three planets (Mercury, Venus, and Earth) around the sun. We want to give an impression of apparent distances and circulation periods. So we approximate the real cosmic distances and scale them down to Open Geometry size.[1]

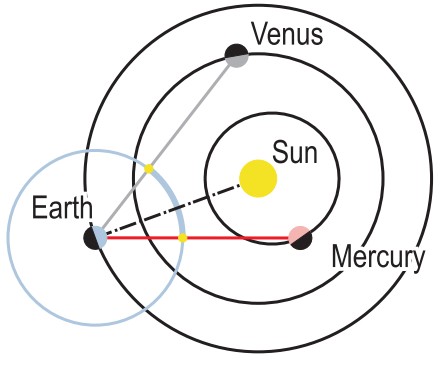

FIGURE 2.1. Output of `"three_planets.cpp"`.

The path curves of the planets are ellipses with the sun as common focal point. Their supporting planes are almost identical (i.e., they have rather small intersection angles). The eccentricity (a measure of the flatness of an ellipse) of most planet paths is close to zero. For this reason, we will approximate the path ellipses of Venus and Earth by circles around the sun. Their radii are 108 million kilometers and 150 million kilometers, respectively. Only the path curve of Mercury is visibly an ellipse. Its distance to the sun varies between 46 million kilometers and 70 million kilometers.

[1] As a general rule, you should scale all your objects to a size such that they fit in a standard Open Geometry window. The alternative of zooming in or zooming out is possible as well, but may disturb the z-buffering algorithm in 3D applications.

The velocity of the planets is determined by KEPLER's second law and is not constant (compare Example 2.2). In order to simplify things, we will, however, assume a constant angular velocity for the revolution around the sun. The circulation periods of Mercury, Venus, and Earth are 88, 225, and 365 Earth days, respectively.

The global constants in `"three_planets.cpp"` are as follows:

Listing from `"three_planets.cpp"`:

```
const P2d Sun = Origin2d;

const Real Factor = 0.05;

const Real VenusDistance = 108 * Factor;
const Real EarthDistance = 150 * Factor;

const Real EarthVelocity = 0.25;
const Real VenusVelocity = EarthVelocity / 225 * 365;
const Real MercuryVelocity = EarthVelocity / 88 * 365;
```

Factor is the zoom factor that adjusts the image to our computer screen. It is not absolutely necessary to use a constant of our own for the sun (we could use *Origin2d* instead), but it will make the code more readable and lucid. Furthermore, we need three global variables for the animation of the scene: Two points V and E (the centers of Venus and Earth) and a straight line SunMercury connecting the sun with the current position of Mercury. Revolving these elements around the sun will give the animation. We define some starting configuration in Init( ):

Listing from `"three_planets.cpp"`:

```
void Scene::Init( )
{
  V.Def( VenusDistance, 0 );
  V.Rotate( Sun, 100 );
  E.Def( EarthDistance, 0 );
  E.Rotate( Sun, 200 );
  SunMercury.Def( Sun, P2d( 17, -24 ) );
  AllowRestart( );
}
```

and continue building up the scene in Draw( ). The path curves of Venus and Earth are circles of given radius with the common center Sun. Apart from that, we draw a sky circle, i.e., a circle around E. It will serve as a projection line for the positions of Venus and Mercury. In our program the planets themselves are small circles. Here, we cannot take into account cosmic dimensions: even a single pixel would be much too large. Apart from that, we draw a black semicircle for the dark side of the planets. The entire code for Venus reads thus:

Listing from `"three_planets.cpp"`:

```
const Real rad_v = 0.5;
Circ2d venus;
venus.Def( Gray, V, rad_v, FILLED );

V2d dir;
dir.Def( V.x, V.y );
dir.Normalize( );
dir = V2d( dir.y, -dir.x );
P2d X;
X = V + rad_v * dir;
Sector2d venus_shadow;
venus_shadow.Def( Black, X, V, 180, 10, FILLED );
```

The semicircle is a *Sector2d* object with a central angle of 180° and 10 points on its circumference. We determine its starting point by rotating the radius vector of the Venus circle through 90°.

The path ellipse of Mercury is initialized as follows:

Listing from `"three_planets.cpp"`:

```
Conic path_of_mercury;
P2d A, B, C;
A.Def( -46, 0 );
B.Def( 70, 0 );
C.Def( 12, 56.745 );
A *= Factor;
B *= Factor;
C *= Factor;
path_of_mercury.Def( Black, 50, A, B, C, Sun );
path_of_mercury.Draw( THIN );
```

From the input data mentioned above we compute three points A, B, and C on the ellipse's axes. We scale them with the global zoom factor and define a conic by these three points and one focal point (Sun). In order to draw the planet itself, we intersect the straight line SunMercury with the path ellipse:

Listing from "three_planets.cpp":

```
int n;
P2d M1, M2;
n = path_of_mercury.SectionWithStraightLine( SunMercury, M1, M2 );
if ( SunMercury.GetParameter( M1 ) < 0 )
    M1 = M2;
Circ2d mercury;
const Real rad_m = 0.5;
mercury.Def( Red, M1, rad_m, FILLED );
```

We need not worry about the number n of intersection points. As SunMercury runs through one focal point of the ellipse, there will always be two of them. Taking the intersection point on one half-ray avoids surprises that stem from the order of the two solutions M1, M2. The dark side of Mercury can be determined in the same way as above.

Now we intersect the straight lines connecting Earth with Venus and Mercury, respectively, with the sky circle. We take the intersection points closer to Venus and Mercury, respectively, to define an arc on the sky circle that gives a good impression of the apparent distance of Venus and Mercury for a viewer on Earth.

Listing from "three_planets.cpp":

```
StrL2d v;
v.Def( E, V );
P2d Q1, Q2;
n = sky_circle.SectionWithStraightLine( v, Q1, Q2 );
if ( Q2.Distance( V ) < Q1.Distance( V ) )
    Q1 = Q2;

StrL2d m;
m.Def( E, M1 );
P2d R1, R2;
n = sky_circle.SectionWithStraightLine( m, R1, R2 );
if ( R2.Distance( M1 ) < R1.Distance( M1 ) )
    R1 = R2;

Sector2d arc;
arc.Def( LightBlue, Q1, E, R1, 15, EMPTY );
arc.Draw( true, THICK );

StraightLine2d( Gray, E, V, THIN );
StraightLine2d( Red, E, M1, THIN );
if ( FrameNum( ) % 10 < 7 )
{
    Q1.Mark( Yellow, 0.15 );
    R1.Mark( Yellow, 0.15 );
}
```

We mark the intersection points only in certain frames to get a twinkling effect (although planets do not really twinkle). In the remaining part of Draw( ) we simply draw the planets and their dark sides (Figure 2.1). ♢

The next example, too, stems from the field of astronomy. We parameterize an ellipse according to KEPLER's laws.

Example 2.2. Kepler's law
The planets of our solar system orbit around the sun according to KEPLER's first and second laws (Figure 2.2):

1. The path curve of a planet P around the sun S is an ellipse with focal point S.
2. During equal time intervals Δ the line segment PS sweeps sectors of equal area.

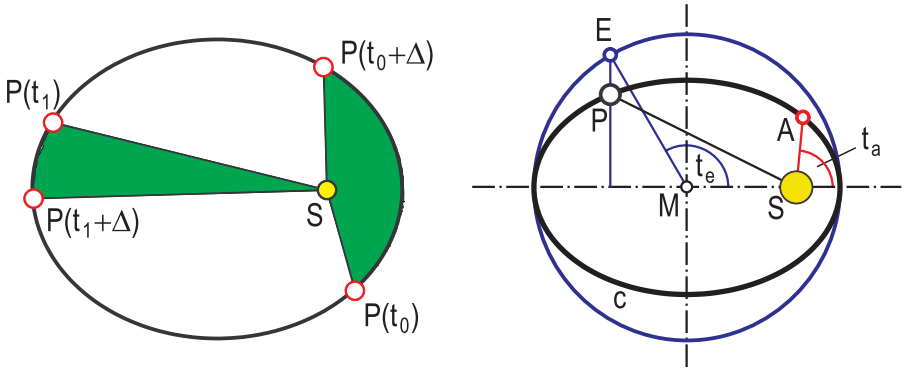

FIGURE 2.2. KEPLER's first and second laws (left). The true planet P, the average planet A, and the eccentric planet E (right).

These two laws are enough to parameterize the path ellipse with the time parameter t. It is, however, not possible to write down the parameterized equation using only elementary functions. Still, we can simulate the motion in an OPEN GEOMETRY program (`"keplers_law.cpp"`).

In order to describe the planet's motion, KEPLER introduced two "virtual planets": The *average planet* A and the *eccentric planet* E. The true planet's path curve is an ellipse c with midpoint M and semiaxes a and b. The path of A is the same ellipse c. It revolves around the sun S with constant angular velocity so that it has the same circulation period as P. The eccentric planet's path is a circle around M with radius a. It moves in a way that the straight line EP is always parallel to the ellipse's minor axis (Figure 2.2).

We denote the angle $\angle SME$ by t_e (*eccentric amplitude*) and the angle between the ellipse's major axis and SA by t_a (*average amplitude*). Now the KEPLER equation states the relation

$$t_e - \varepsilon \sin t_e - t_a = 0, \tag{1}$$

where ε is the numeric eccentricity $\sqrt{a^2 - b^2}/a$ of c. In order to find the position $P(t)$ of the planet, we compute the position $A(t)$ of the average planet and the angle t_a, solve equation (1) for t_e, find $E(t)$ and subsequently $P(t)$. This seems to be easy, but unfortunately, there exists no elementary solution to (1). We have to use iteration methods to determine the roots.

For programming purposes, it helps considerably to see that (1) has exactly one root for any fixed real t_a: The left-hand side of the equation takes positive and negative values, and it is strictly monotonic (its derivative with respect to t_e is always positive).

In `"keplers_law.cpp"` we decided on the following strategy: We compute a number of positions of the planets before Draw( ) and store them in an array. In Draw( ) we do not perform any computation but simply mark the correct points according to the frame number we use. Besides speeding up the animation a little, this yields the same computation time for each new frame. As a consequence, the animation will be smoother.

In the preamble we decide on the number of points and reserve the required memory for the average planet, eccentric planet, and true planet:

Listing from `"keplers_law.cpp"`:

```
const int N = 150;
const int N2 = 2 * N;
P2d APlanet[N2], EPlanet[N2], TPlanet[N2];
```

Then we write a function returning the value of the KEPLER equation (1):

Listing from `"keplers_law.cpp"`:

```
Real TA;
Real KeplerFunction( Real t )
{
    return t - E / A * sin( t ) - TA;
}
```

Note that this function actually depends on two input parameters: the real t and the global variable TA, where the current parameter value of the average planet is stored. E is the linear eccentricity $\sqrt{a^2 - b^2}$ of c. Using the average amplitude t_a and the eccentric amplitude t_e, we can parameterize the path curves of the planets as follows:

Listing from "keplers_law.cpp":

```
P2d AveragePlanet( Real ta )
{
    return Sun + B * B / ( E * cos( ta ) + A ) *
           V2d( cos( ta ), sin( ta ) );
}
P2d EccentricPlanet( Real te )
{
    return P2d( A * cos( te ), A * sin( te ) );
}
P2d TruePlanet( Real te )
{
    return P2d( A * cos( te ), B * sin( te ) );
}
```

In Init( ) we compute the path points in a simple **for** loop. The Open Geometry class *Function* provides a root-finding algorithm for that purpose. We need not bother about the number of solutions; there exists exactly one. Furthermore, we can gain half the points by a simple reflection on the x-axis.

Listing from "keplers_law.cpp":

```
int i;
Real delta = PI / ( N - 1 );
for ( i = 0, TA = 0.001; i < N; i++, TA += delta )
{
    APlanet[i] = AveragePlanet( TA );
    Function eccentric( KeplerFunction );
    eccentric.CalcZeros( -3.3, 3.3, 50, 1e-3 );
    EPlanet[i] = EccentricPlanet( eccentric.Solution( 1 ) );
    TPlanet[i] = TruePlanet( eccentric.Solution( 1 ) );
    APlanet[2*N-1-i].Def( APlanet[i].x, -APlanet[i].y );
    EPlanet[2*N-1-i].Def( EPlanet[i].x, -EPlanet[i].y );
    TPlanet[2*N-1-i].Def( TPlanet[i].x, -TPlanet[i].y );
}
```

The rest is clear. In Draw( ) we mark the planets and draw their path curves. The variable Index indicates the current position. It is initialized with zero and changed in Animate( ):

Listing from "keplers_law.cpp":

```
void Scene::Animate( )
{
   Index++;
   Index = Index % N2;
}
```

♢

Open Geometry is a good tool for illustrating theorems of elementary plane geometry. As an example, we choose a recent theorem that was published as a question by J. Fukuta in 1996 and proved by Z. Čerin in 1998 ([5]). It states that applying certain operations to a triangle will always result in a regular hexagon (Figure 2.3).

Example 2.3. Fukuta's theorem
We start with an arbitrary triangle and divide each side into two fixed affine ratios λ_1 and λ_2. These new points form a hexagon. To each side of the hexagon we add a third point to form an equilateral triangle. Three consecutive of these points define a triangle and the barycenters of these triangles lie on a *regular hexagon*. Furthermore, the barycenters of the initial triangle and the final hexagon are identical.

Now, what can Open Geometry do with this theorem? Of course, we can implement the construction of the regular hexagon ("fukuta1.cpp"). For frequent tasks we introduce three auxiliary functions:

Listing from "fukuta1.cpp":

```
P2d AffComb( P2d &A, P2d &B, Real lambda )
{
   return P2d( ( 1 − lambda ) * A.x + lambda * B.x,
               ( 1 − lambda ) * A.y + lambda * B.y );
}

void MakeTriangle( const P2d &X, const P2d &Y, Color col,
                   Poly2d &poly )
{
   P2d Z( 0.5 * X.x + 0.5 * Y.x + sqrt( 3 ) / 2 * ( Y.y − X.y ),
          0.5 * X.y + 0.5 * Y.y + sqrt( 3 ) / 2 * ( X.x − Y.x ) );
```

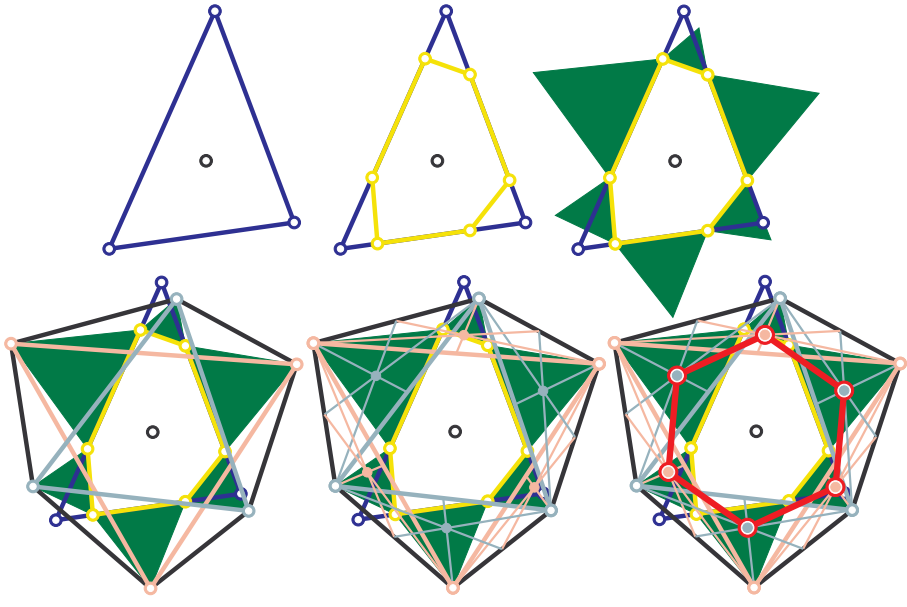

FIGURE 2.3. FUKUTA's theorem.

```
    poly.Def( col, 3, FILLED );
    poly[1] = X;
    poly[2] = Y;
    poly[3] = Z;
}

void DrawMedians( P2d &A, P2d &B, P2d &C, Color c,
                  ThinOrThick thick )
{
    P2d AM = 0.5 * P2d( B.x + C.x, B.y + C.y );
    P2d BM = 0.5 * P2d( C.x + A.x, C.y + A.y );
    P2d CM = 0.5 * P2d( A.x + B.x, A.y + B.y );
    StraightLine2d( c, A, AM, thick );
    StraightLine2d( c, B, BM, thick );
    StraightLine2d( c, C, CM, thick );
}
```

AffComb(...) gives the intermediate point of a line segment that corresponds to a given affine ratio lambda, MakeTriangle(...) defines a regular triangle with two given vertices as *Poly2d* object, and DrawMedians(...) draws the medians of a triangle.

With the help of these functions it is easy to construct and display all relevant objects of FUKUTA's theorem. We draw the initial triangle in the first frame, add the first hexagon in the second frame, the regular triangles in the third frame, and so on. This makes the theorem's contents much more lucid than one single picture. By switching from frame to frame you will be able to watch the development of the regular hexagon.

In order to get an attractive image, it is important to choose the line styles and shading options with care. For this reason, we did not, e.g., shade the triangles whose barycenters lie on the regular hexagon. Furthermore, it is important to mark the points after drawing the straight lines passing through them. Otherwise, they would be covered by the other lines.[2]. The actual implementation of the drawing routines is rather lengthy. We do not display it here but you can find them in `"fukuta1.cpp"`

We can extend the previous program and write an OPEN GEOMETRY *animation* of FUKUTA's theorem. In order to do this, we have only to define the input data (i.e., the first triangle and the two affine ratios) globally, change their values in Animate( ), and draw the whole picture in every single frame (`"fukuta2.cpp"`). The animation comes from redefining the vertices of the initial triangle and the affine ratios in Animate( ). Run `"fukuta2.cpp"` and watch it! ♢

Now we present the first example using a very fundamental 2D class: a parameterized plane curve (**class** *ParamCurve2d*). It is among the most frequently used classes in OPEN GEOMETRY. If you want to display a curve of your own, we recommend starting with the templet `"paramcurve2d.cpp"` from `"USER/TEMPLETS/"`.

Example 2.4. The quadratrix
There are three classic geometric problems that the ancient Greeks tackled but could not solve. For many centuries thereafter, mathematicians tried to solve them, but it was not until the development of modern algebra that they were finally proved to be unsolvable. These classic problems are:

1. the squaring of a circle;
2. the trisection of an arbitrary angle;
3. the doubling of the volume of a cube.

[2]In 2D mode, the drawing order of the objects determines their visibility relations.

The Greeks were interested only in geometric solutions that use no other means than straighedge and compass. Ultimately, everything ends up in the question of whether certain irrational numbers (e.g., π or $\sqrt[3]{2}$) are "constructible" or not (today we know that they are not).

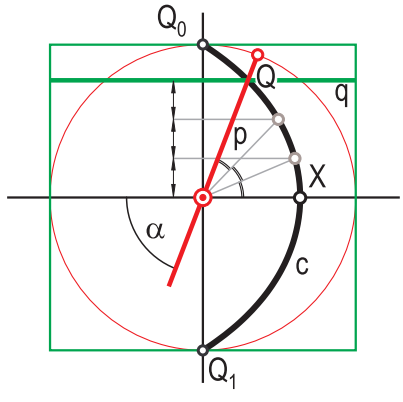

FIGURE 2.4. Generation of the quadratrix.

The squaring of the circle and the trisection of an arbitrary angle are related to a special plane curve c. Therefore, c is called either a *quadratrix* or *trisectrix*.

We can generate the quadratrix by very simple kinematic methods (Figure 2.4). Suppose that p and q are two straight lines through a point Q_0 with coordinates $(0, 1)$. The line p runs through the origin O of our coordinate system; q is parallel to the x-axis.

We rotate p about O and translate q in the direction of the y-axis. For both motions we assume constant velocities so that p and q intersect in $Q_1(0, -1)$ after a half turn of p. The quadratrix is defined as the locus of all intersection points of p and q during these motions.

Of course, c is not an algebraic curve. An arbitrary line through O intersects c in infinitely many points. The first intersection point X of x and c is of special interest. Using the parameterized equation

$$c\colon \vec{x}(t) = (1-t)\begin{pmatrix} \tan\frac{\pi}{2}t \\ 1 \end{pmatrix}$$

of c, we obtain $X = \lim_{t\to 1} \vec{x}(t) = (2/\pi, 0)^t$. Thus, c can be used for a graphic construction of a transcendental number related to π. If it could be constructed exactly (i.e., if there existed a finite algorithm to determine it with the help of straightedge and compass only), one would have a solution to the squaring of a circle.

The trisection problem of an angle $\alpha = \pi/2t$ can also be solved with the help of the quadratrix. In order to divide α in three equal parts, we draw three horizontal

lines at distance $it/3$ from the origin and intersect them with c ($i = 1, 2, 3$; Figure 2.4).

In "quadratrix.cpp" we illustrate these properties of the quadratrix. We implement it as a parameterized curve:

Listing from "quadratrix.cpp":

```
class MyQuadratrix: public ParamCurve2d
{
public:
   P2d CurvePoint( Real t )
   {
      if ( fabs( 1 - t ) > 0.005 )
         return P2d( S * ( 1 - t ) * tan( 0.5 * PI * t ), S * ( 1 - t ) );
      else
         return P2d( 2 * S / PI, 0 );
   }
};
MyQuadratrix Quadratrix;
```

The value S is a globally defined scale factor that we use to fill a standard OPEN GEOMETRY drawing screen. The CurvePoint(...) function takes care of the special case $t = 1$ where $\vec{x}(t)$ is undefined. For the animation we use a global point P and a global straight line p. They are rotated and translated by constant increments:

Listing from "quadratrix.cpp":

```
const Real Delta = -0.75;
V2d TranslVec( 0, 2 * S * Delta / 180 );
```

In order to point out the use of c for the trisection of an arbitrary angle, we use two points R1 and R3. These points belong to the parameter values $t_0/3$ and $2t_0/3$, respectively, if Q = OP ∩ p is the curve point $\vec{x}(t_0)$. In "quadratrix.cpp" we have to take care of the scale factor S and determine R1 and R3 as follows:

Listing from "quadratrix.cpp":

```
if ( fabs( P.y ) > 0.02 )
   Q = p * q;
else
   Q = Quadratrix.CurvePoint( 1 );
P2d R1 = Quadratrix.CurvePoint( 1 - Q.y / S / 3 );
P2d R2 = R1;
R2.Reflect( Yaxis2d );
P2d R3 = Quadratrix.CurvePoint( 1 - 2 * Q.y / S / 3 );
P2d R4 = R3;
R4.Reflect( Yaxis2d );
```

The points R2 and R4 are just symmetric points for obtaining a prettier image. For reasons of symmetry (after all, we are geometers!) we actually print two quadratrices and move p up and down inside a square. Otherwise, p would soon leave the range of your computer screen. ♢

Associated curves

There exist many ways of associating a curve c^* with a given plane curve c: $\vec{x} = \vec{x}(u)$ (compare, e.g., `http://www.xahlee.org` or related sites on the internet). In the following we will present some of them. In Open Geometry 2.0 they are all implemented as methods of *ParamCurve2d*. Thus, you can display these special curves almost immediately on the screen.

Example 2.5. The evolute
An *osculating circle* o of c is a circle that intersects c in three neighboring points. The locus of all centers of osculating circles is called the *evolute* e of c. Alternatively, e can be defined as the *hull curve of all normals* n *of* c. Figure 2.5 shows the evolute of an ellipse. We have used a slightly modified version of "get_evolute.cpp" to produce this picture.

The method GetEvolute(...) is a member of *ParamCurve2d*. Its implementation in "f.cpp" reads as follows:

Listing from "C/f.cpp":

```
void ParamCurve2d::GetEvolute( Real u1, Real u2,
   L2d &evolute )
{
   int n = evolute.Size( );
   const Real eps = 1e-5;
   Real u = u1;
```

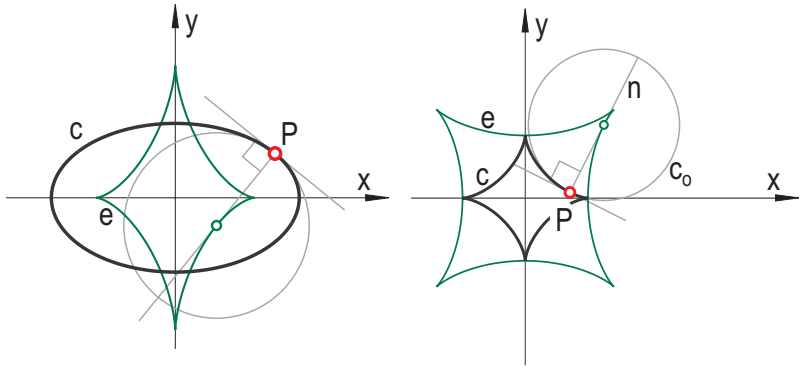

FIGURE 2.5. Evolutes of an ellipse (left) and an astroid (right).

```
    Real delta = ( u2 − u1 ) / ( n − 1 );
    for ( int i = 1; i <= n; i++, u += delta )
      evolute[i] = Normal( u − eps ) ∗ Normal( u + eps );
}
```

It needs three arguments: The two real numbers u1 and u2 define the parameter interval $[u1, u2]$ in which the evolute is drawn; the points of the evolute e are stored in the *L2d* object evolute.

The code itself is very simple. We get the number n of points in the *L2d* object evolute, divide the parameter interval $[u1, u2]$ into $n - 1$ parts of equal length, and intersect two "neighboring" normals[3] of c that correspond to the partition points.[4]

By the way, if you want to use all the methods of *ParamCurve2d* with the evolute, you can define a second parameterized curve in your OPEN GEOMETRY program. The virtual function CurvePoint(...) has to be more or less the same as the code in GetEvolute(...). You have only to specify the curve name before calling any methods of the base curve (e.g., MyCurve.CurvePoint(...) instead of CurvePoint(...)). The same holds for all special curves in the following text. ♢

Example 2.6. The catacaustic
Suppose now that c is a *reflecting* curve: A ray of light intersecting c is reflected according to the law of reflection: *The normal n of c is bisectrix of the incoming and outgoing rays* (see Figure 2.6).

[3]By "neighboring" we mean two normals that are *sufficiently close*; the real constant eps serves to define this term.

[4]This is more or less a "geometric construction" of the evolute. OPEN GEOMETRY's philosophy aims at avoiding complicated computations. Instead, the user should immediately transfer geometric ideas to the program file.

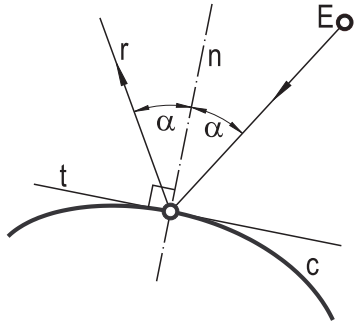

FIGURE 2.6. The law of reflection.

If you reflect the lines of a pencil with vertex E on c, the reflected lines are all tangent to a certain curve c_c – the *catacaustic of c with respect to the pole E*. In order to implement the method GetCata(...) of the class *ParamCurve2d* in Open Geometry, we basically did the same as in the previous example: We intersect two "neighboring" reflected rays:

Listing from `"C/f.cpp"`:

```
void ParamCurve2d::GetCata( P2d &Pole, Real u1, Real u2,
    L2d &catacaustic )
{
    int n = catacaustic.Size( );
    const Real eps = 1e-5;
    Real u = u1;
    Real delta = ( u2 - u1 ) / ( n - 1 );
    for ( int i = 1; i <= n; i++, u += delta )
    {
        Real u_a = u - eps, u_b = u + eps;
        P2d A = CurvePoint( u_a );
        P2d B = CurvePoint( u_b );
        StrL2d a( Pole, A );
        StrL2d b( Pole, B );
        a.Reflect( Tangent( u_a ) );
        b.Reflect( Tangent( u_b ) );
        catacaustic[i] = a * b;
    }
}
```

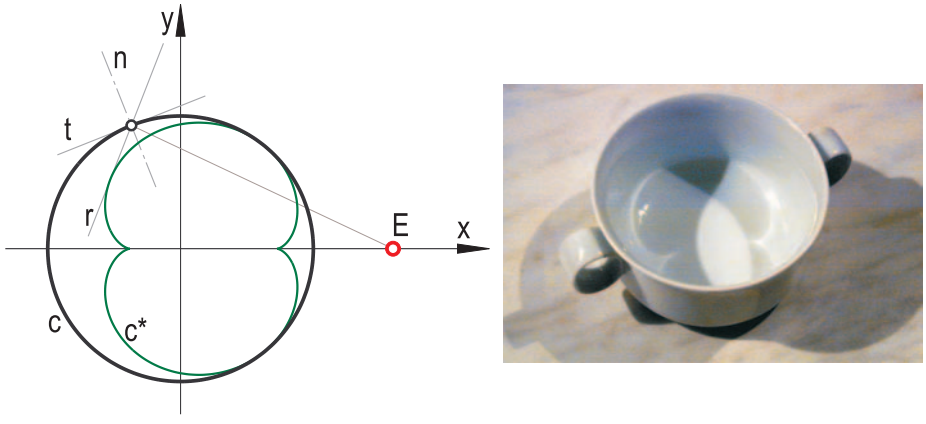

FIGURE 2.7. The catacaustic c_c of a circle c with respect to the pole E. On the right-hand side you can see two "real world" caustics in a cup.

Of course Pole is identical to the pole E, while the meaning of the input parameters u1, u2, and catacaustic is analogous to GetEvolute(...). In order to produce Figure 2.7, we used the program `"get_catacaustic.cpp"`. You can see the catacaustic c_c of a circle c and one reflected ray r being tangent to c_c. More on catacaustics and diacaustics (caustics of refraction) can be found in various places in this book (see Examples 2.32, 2.43, 2.44, 3.25 and 4.12)). ♢

Example 2.7. The pedal curve
Things become easier when the points of the associated curve can be constructed explicitly (and not by intersecting two "neighboring" tangents). This is the case with the *pedal curve* c_p. The points of c_p are the pedal points of the pole E on all the tangents of c (Figure 2.8). Since the OPEN GEOMETRY class *StrL2d* provides the method NormalProjectionOfPoint(...), the code of GetPedal(...) is really simple:

Listing from `"C/f.cpp"`:

```
void ParamCurve2d::GetPedal( P2d &Pole, Real u1, Real u2, L2d &pedal )
{
   int n = pedal.Size( );
   Real u = u1;
   Real delta = ( u2 - u1 ) / ( n - 1 );
   for ( int i = 1; i <= n; i++, u += delta )
      pedal[i] = Tangent( u ).NormalProjectionOfPoint( Pole );
}
```

♢

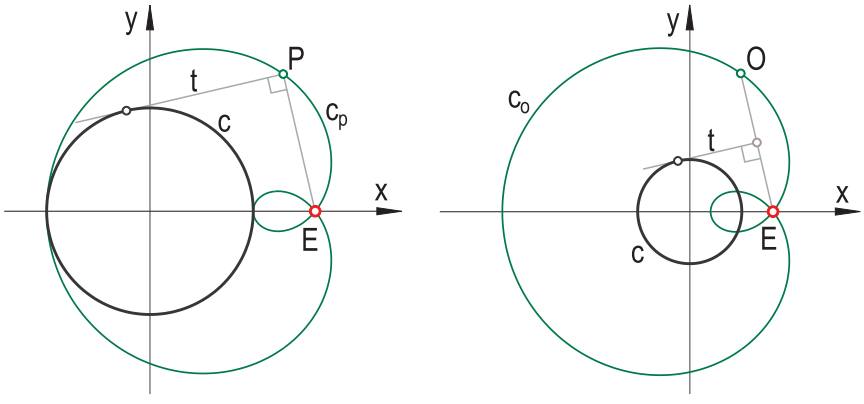

FIGURE 2.8. The pedal curve c_p (left, compare `"get_pedal.cpp"`) and the orthonomic c_o (right, compare `"get_orthonomic.cpp"`) of a circle c. Both curves are limaçons of Pascal.

Example 2.8. The orthonomic

If you reflect E on all tangents of c, you get another special curve: the *orthonomic* c_o *of* c *with respect to* E (Figure 2.8). By definition, c_o is homothetic to the pedal curve c_p with center E and factor 2. Its implementation in OPEN GEOMETRY reads as follows

Listing from `"C/f.cpp"`:

```
void ParamCurve2d::GetOrtho( P2d &Pole, Real u1, Real u2,
    L2d &orthonomic )
{
    int n = orthonomic.Size( );
    Real u = u1;
    Real delta = ( u2 - u1 ) / ( n - 1 );
    for ( int i = 1; i <= n; i++, u += delta )
    {
        P2d P = Pole;
        P.Reflect( Tangent( u ) );
        orthonomic[i] = P;
    }
}
```

In contrast to the method GetPedal(...), we have to take into account that a point will be changed when reflected on a straight line. Therefore, we copy the pole to the point P before passing it as an argument. ♢

Example 2.9. The antipedal curve and the antiorthonomic
Being given a plane curve c, it is easy to find the antipedal a_p and the antiorthonomic a_o (i.e., the curves having c as pedal curve or orthonomic, respectively). Figure 2.9 shows their construction. Their implementation in OPEN GEOMETRY (methods GetAntiPedal(...) and GetAntiOrtho(...)) is just as straightforward, so we will not display the code at this place.

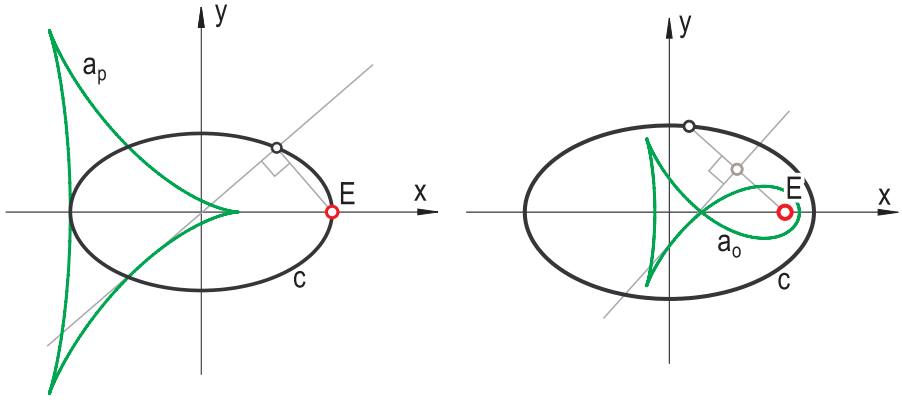

FIGURE 2.9. Antipedal curve (left, compare `"get_anti_pedal.cpp"`) and the antiorthonomic (right, compare `"get_anti_ortho.cpp"`) of an ellipse c.

♢

Now we turn to another class of associated curves: Sometimes, the associated curve depends on an additional parameter. That is, there exists a one-parameter family of associated curves. Our first example of this type is the involute:

Example 2.10. The involute
The involute i of c might as well be called the antievolute; i.e., the evolute of i is c. The involute is, however, not uniquely determined. Each point on c is the start point of its own involute curve, and we need an additional parameter (e.g., the curve parameter u) to specify it.

Figure 2.10 shows the geometric construction of the involute i_P to a point $P = P(u_0) \in c$. If $Q = Q(u)$ is a second point on c, we can find a point I_P of i_P on the tangent t_Q of c in Q. The distance between Q and I_P is equal to the arc length a between P and Q measured on c.

You can see the following properties of an involute curve:

- All tangents of c are perpendicular to any involute of c.
- The involutes belong to a family of offset curves (compare the following example).

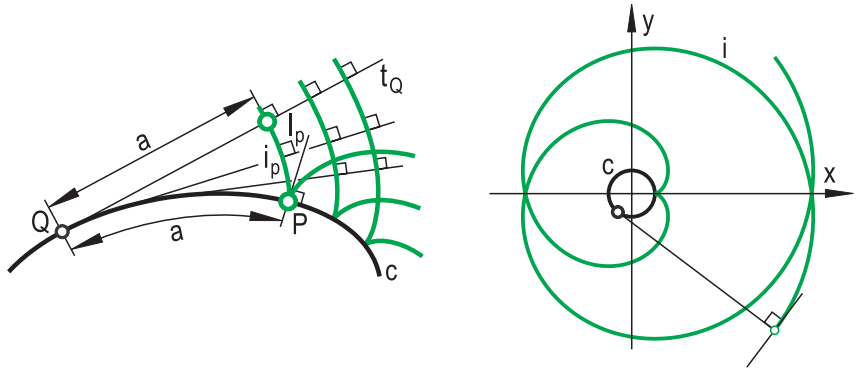

FIGURE 2.10. Involutes of a plane curve c (left) and the involute of a circle (right).

- P is a cusp of i_P and the curve normal of c in P is the tangent of i_P in P.

We implemented a method GetInvolute(...) in OPEN GEOMETRY that reads as follows:

Listing from "C/f.cpp":

```
void ParamCurve2d::GetInvolute( Real param_value, Real u1, Real u2,
    L2d &involute )
{
    int n = involute.Size( );
    const int accuracy = 300;
    Real u = u1;
    Real delta = ( u2 − u1 ) / ( n − 1 );
    for ( int i = 1; i <= n; i++, u += delta )
    {
        StrL2d t = Tangent( u );
        int sign = Signum( param_value − u );
        Real length = sign ∗ ArcLength( param_value, u, accuracy );
        involute[i] = t.InBetweenPoint( length );
    }
}
```

The arguments u1, u2 and involute have the usual meaning. But GetInvolute(...) needs the curve parameter u = param_value as an additional argument to specify the cusp $P = P(u)$ of the involute. As an example we wrote the program "get_involute.cpp". Its output, the involute of a circle c, can be seen on the right-hand side of Figure 2.10. ♢

Example 2.11. Offset curves
The *offset curve* o_d *at distance* d of c is well-known in geometry and computer graphics (d is a real constant). With the help of the unit normal vector $\vec{n}(u)$ it can be parameterized according to

$$o_\mathrm{d} \ldots \vec{y}(u) = \vec{x}(u) + d\vec{n}(u).$$

The point $\vec{y}(u)$ is located on the curve normal through $\vec{x}(u)$. The distance between these two points is always d.

This fixed distance is a parameter to determine the offset curve o_d. In Figure 2.11 two offset curves of an astroid at distance $\pm d$ are displayed (compare `"get_offset.cpp"`).

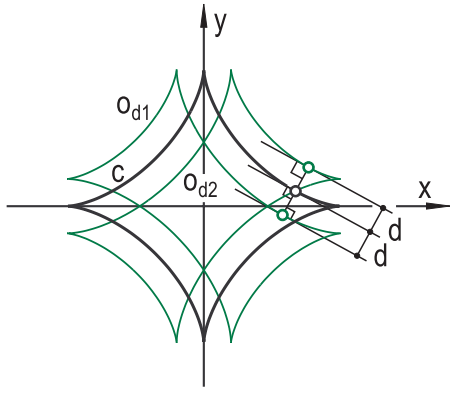

FIGURE 2.11. Two offset curves of an astroid c.

Thus, the method GetOffset(...) again needs four arguments.

Listing from `"C/f.cpp"`:

```
void ParamCurve2d::GetOffset( Real distance, Real u1, Real u2,
    L2d &offset )
```

The parameter to specify the offset curve is the distance distance. Its OPEN GEOMETRY implementation can be found in `"f.cpp"`. ♢

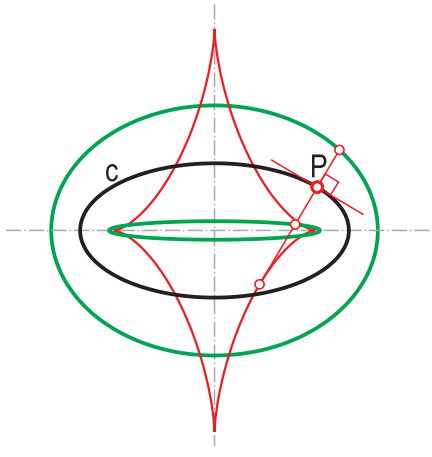

FIGURE 2.12. Two ellipses as general offset curves of an ellipse c (`"general_offset.cpp"`).

Example 2.12. Generalized offset curves
The concept of offset curves can be generalized a little. The distance d on the normals of c may depend on the curve parameter u. We did not implement this special curve as a method of *ParamCurve2d*, but we wrote a sample file `"general_offset.cpp"` to present a surprising example of this curve type.

At first, we define an ellipse as base curve c (in fact, you may use any conic section here). Next, we implement a function OffsetDistance(...):

Listing from `"general_offset.cpp"`:

```
Real OffsetDistance( Real u )
{
    Real rad_of_curvature = Curve.RadiusOfCurvature( u );
    return exp( Log( rad_of_curvature ) / 3 );
}
```

It returns the cube root of the radius of curvature of c in the curve point $P = P(\mathsf{u})$. Using this function we derive an object of type *ParamCurve2d* called GenOff1:

Listing from `"general_offset.cpp"`:

```
class GeneralOffset1: public ParamCurve2d
{
```

```
public:
    P2d CurvePoint( Real u )
    {
        V2d normal = Curve.NormalVector( u );
        return Curve.CurvePoint( u ) + OffsetDistance( u ) * normal;
    }
};
GeneralOffset1 GenOff1;
```

In the same way, we define a parameterized curve GenOff2 using the negative offset distance. The general offset curves are then ellipses themselves (Figure 2.12; for a proof see [9]). In fact, we can even take an arbitrary distance function proportional to OffsetDistance(u) and still will get an ellipse. ♢

Families of curves

A single curve is a very common object of geometry. But sometimes this is not enough, and single- or multiple-parametric families of curves are at the center of interest. In this chapter we present three examples of this.

Example 2.13. Pencil of circles
We start with perhaps the most famous family of curves: the pencil of circles (`"pencil_of_circles.cpp"`). It consists of all circles through two given points A and B. For the time being, we assume that A and B are real points. We assign the coordinate vectors $(-\mathsf{R}, 0)^t$ and $(\mathsf{R}, 0)^t$, respectively, to them.

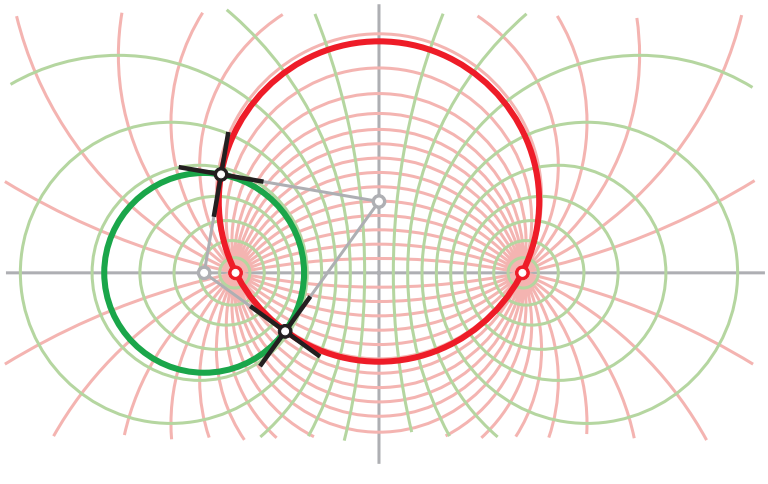

FIGURE 2.13. A pencil of circles and the pencil of its orthogonal circles.

In our program we define those points as global constants, with C0 the circle with diameter AB. It is now very easy to draw members of the pencil of circles $\mathcal{P}_1$.

We decide on the total number CircNum of circles and on a certain "distance" Delta. Then we write in Draw( ):

Listing from "pencil_of_circles.cpp":

```
Circ2d circ;
P2d C, D;
int i;
Real t;

t = -0.5 * Delta * CircNum;
for ( i = 0; i < CircNum; i++, t += Delta )
{
    C.Def( 0, t );
    circ.Def( LightRed, A, B, C );
    circ.Draw( THIN );
}

t = -0.5 * Delta * CircNum;
for( i = 0; i < CircNum; i++, t += Delta )
{
    C.Def( t, 0 );
    D = C;
    C0.InvertPoint( D );
    D.Def( 0.5 * ( C.x + D.x ), 0.5 * ( C.y + D.y ) );
    circ.Def( LightGreen, D, D.Distance( C ) );
    circ.Draw( THIN );
}
```

We define a few local variables at the top. In the first loop we do nothing but vary the point C on the y-axis and draw the circle through A, B, and C in light red. By the way, the global variable Delta has the value $2R/$CircNum. This is just enough to ensure that no circle is drawn twice.

The second loop looks more interesting. There, we vary C on the x-axis, determine its inversion D at the fixed circle C0, and draw the circle with diameter CD in light green. The circles of the second loop belong to a pencil $\mathcal{P}_2$ of circles as well. The base points are, however, imaginary. Their coordinates are $(0, \pm 2iR)^t$.

The most interesting property of $\mathcal{P}_2$ is the following: *The circles of* $\mathcal{P}_2$ *and* $\mathcal{P}_1$ *intersect orthogonally.* In order to illustrate this, we draw two circles $c_1 \in \mathcal{P}_1$ and $c_2 \in \mathcal{P}_2$ in medium thickness and get their intersection points:

Listing from "pencil_of_circles.cpp":

```
t = X.Next( );
C.Def( t, 0 );
D = C;
C0.InvertPoint( D );
D.Def( 0.5 * ( C.x + D.x ), 0.5 * ( C.y + D.y ) );
GreenCircle.Def( Green, D, D.Distance( C ) );
GreenCircle.Draw( MEDIUM );

t = Y.Next( );
C.Def( 0, t );
RedCircle.Def( Red, A, B, C );
RedCircle.Draw( MEDIUM );

i = RedCircle.SectionWithCircle( GreenCircle, C, D );
```

Here X and Y are objects of type *PulsingReal*. This type is very useful for small animations as in the present example. In the following part, we draw the normals of each circle in the intersection points and mark all relevant points. We need not worry about the number of intersection points: Any two circles from $\mathcal{P}_1$ and $\mathcal{P}_2$ have exactly two points in common. Figure 2.13 shows the output of "pencil_of_circles.cpp". ♢

The circles of the previous example form a net of curves that intersect orthogonally. There exists another well-known net of that type: The net of *confocal conics*.

Example 2.14. Confocal conics
Given two points $\mathsf{F1}, \mathsf{F2} \in \mathbb{R}^2$, there exists a one-parametric family $\mathcal{E}$ of ellipses having F1 and F2 as focal points. Every point of $\mathbb{R}^2$ that is not situated on the straight line F1F2 lies on exactly one ellipse from $\mathcal{E}$. Furthermore, there exists a one-parametric family $\mathcal{H}$ of hyperbolas with the analogous properties.

In "confocal_conics.cpp" we display both families, $\mathcal{E}$ and $\mathcal{H}$. We define the focal points F1, F2 together with their half-distance E (eccentricity) globally, decide on a certain integer ConicNum, and reserve memory for ConicNum ellipses and hyperbolas that will be displayed (in red and green, respectively):

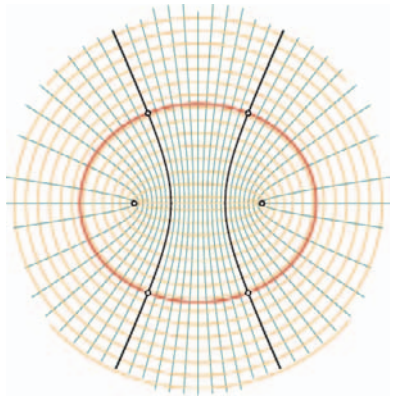

FIGURE 2.14. Families of confocal ellipses and hyperbolas.

Listing from `"confocal_conics.cpp"`:

```
const Real E = 5;
const P2d F1( -E, 0 ), F2( E, 0 );

const int ConicNum = 15;
Conic Ell[ConicNum], Hyp[ConicNum];

Conic RedConic, GreenConic;
PulsingReal X, Y;
```

The conics RedConic and GreenConic will be used for an animation. Most of these global variables are initialized in Init( ):

Listing from `"confocal_conics.cpp"`:

```
void Scene::Init( )
{
   // define background hyperbolas
   int i, j = ConicNum - 2;
   Real delta = E / ( ConicNum - 1 );
   Real t = -E + delta;
   for ( i = 0; i < j; i++, t += delta )
      Hyp[i].Def( LightGreen, 200, P2d( t, 0 ),
         F1, F2, HYPERBOLA );
   Hyp[ConicNum-2].Def( LightGreen, 200, P2d( 0.05 - E, 0 ),
        F1, F2, HYPERBOLA );
```

```
    Hyp[ConicNum−1].Def( LightGreen, 200, P2d( E − 0.05, 0 ),
        F1, F2, HYPERBOLA );

    // define background ellipses
    delta = 1;
    t = 0.5 ∗ delta;
    for ( i = 0; i < ConicNum; i++, t += delta )
      Ell[i].Def( LightRed, 200, P2d( 0, t ), F1, F2, ELLIPSE );

    X.Def( E ∗ 0.11973, 0.05, −E + 0.0001,
      E − 0.0001, HARMONIC );
    Y.Def( E ∗ 1.22495, 0.05, 0.05, 10, LINEAR );

    AllowRestart( );
}
```

Of course, the appropriate defining method for the conics uses one conic point and the two common focal points. Additionally, we must specify the respective conic type (*ELLIPSE* or *HYPERBOLA*) in order to avoid ambiguities. In Draw( ) we display these conics. Additionally, we draw the degenerate members (straight line segments) of each of the families $\mathcal{E}$ and $\mathcal{H}$, define two special conics, and mark their intersection points:

Listing from `"confocal_conics.cpp"`:

```
Yaxis2d.Draw( LightGreen, −15, 15, THIN );
Xaxis2d.Draw( LightGreen, −15, −E, THIN );
Xaxis2d.Draw( LightGreen, E, 15, THIN );
Xaxis2d.Draw( LightRed, −E, E, THIN );

RedConic.Def( Red, 150, P2d( 0, Y.Next( ) ),
    F1, F2, ELLIPSE );
GreenConic.Def( Green, 150, P2d( X.Next( ), 0 ),
    F1, F2, HYPERBOLA );
RedConic.Draw( THICK );
GreenConic.Draw( THICK, 15 );

const Real x = RedConic.DistMA( ) ∗ GreenConic.DistMA( ) / E;
const Real y = RedConic.DistMC( ) ∗ GreenConic.DistMC( ) / E;
P2d( x, y ).Mark( Black, 0.2, 0.1 );
P2d( x, −y ).Mark( Black, 0.2, 0.1 );
P2d( −x, y ).Mark( Black, 0.2, 0.1 );
P2d( −x, −y ).Mark( Black, 0.2, 0.1 );
```

Note that we deliberately avoid the SectionWithConic(...) method of the class *Conic* for the computation of the intersection points (last six lines of the above listing). This method finds the intersection point by solving an algebraic equation of order four. Our calculation is simpler. Additionally, we avoid numerical problems that arise from the fact that the conic axes are parallel to the coordinate axes x and y.[5] ♢

Example 2.15. Cassini's oval
In the preceding example we have considered the set of conics with common focal points F_1 and F_2. Each such conic consists of all points $X \in \mathbb{R}^2$ satisfying $\overline{XF_1} \pm \overline{XF_2} = 2a$. Cassini's oval o (G.D. Cassini, 1625–1712) is defined in a similar way:

$$o := \{X \in \mathbb{R}^2 \mid \overline{XF_1} \cdot \overline{XF_2} = a^2\}, \quad a \in \mathbb{R} = \text{const.}$$

Actually, there exist many different forms of Cassini's ovals. Their shape depends on the ratio $e : a$, where $2e$ is the distance between F_1 and F_2. In any case, o is a quartic curve. Its implicit equations in Cartesian coordinates and polar coordinates read

$$o \ldots (x^2 + y^2)^2 - 2e^2(x^2 - y^2) = a^4 - e^4 \tag{2}$$

and

$$o \ldots r^2 = e^2 \cos 2\varphi \pm \sqrt{a^4 - e^4 \sin^2 2\varphi}, \tag{3}$$

respectively. For $a \geq e$ the curve is connected; for $a \geq \sqrt{2}e$ it is even convex. For $a = e$ the origin is a double point, and Cassini's oval is identical to Bernoulli's lemniscate. Finally, for $a \leq e$ the curve splits into two parts symmetric to the y-axis (compare Figure 2.15).

Both equations (2) and (3) are not too good for an Open Geometry visualization. The main problem consists in the different shapes of the curve. There exists no proper parameterized equation of o covering all cases. Thus, we will try a different approach to the problem. Instead of drawing line segments (as with an *L2d*, *PathCurve2d*, or *ParamCurve2d* object), we will mark a few hundred curve points. This turns out to be much faster in an animation that shows the transition between different shapes of the curve.

In `"cassini.cpp"` we use three global constants: E (the "focal distance"), PointNum (the number of points to be drawn) and Thickness (a real value that determines the radius of the circles that mark the points). Our only global variable will be the pulsing real A that occurs in equations (2) and (3). It is defined in Init( ):

[5] This leads to zeros of multiplicity $\mu > 1$ of the algebraic of order four. Our current root-solving algorithm is not very stable in this case.

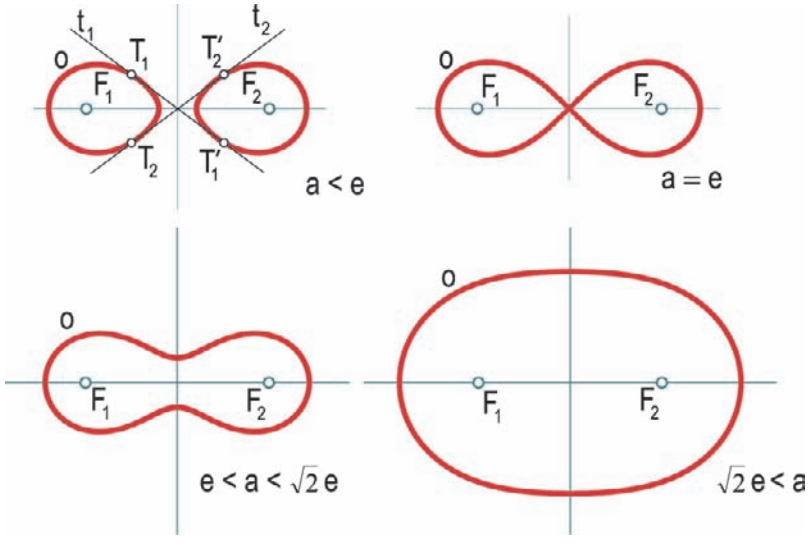

FIGURE 2.15. Different shapes of Cassini's oval.

Listing from `"cassini.cpp"`:

```
A.Def( E − 0.059, 0.07, −2.5 ∗ E, 2.5 ∗ E, HARMONIC );
```

The weird initial value ensures that o will be more or less a lemniscate in the first frame. Our strategy is now the following:

1. We start with a straight line $s(\varphi)$ of direction $(\cos\varphi, \sin\varphi)^t$ through the origin.
2. We intersect $s(\varphi)$ with o. The polar coordinates of the up to four solution points can be computed from (3).
3. We mark the solution points, rotate $s(\varphi)$ about the origin through a given small angle δ, and start the same procedure again.

This is done in Draw( ):

Listing from `"cassini.cpp"`:

```
const Real E2 = E ∗ E;
const Real E4 = E2 ∗ E2;
const Real a = A.Next( );
```

```
const Real a4 = a * a * a * a;

Real phi, delta = 2 * PI / PointNum, c2, x, r, c, s;

for ( phi = 0; phi <= 2 * PI; phi += delta )
{
   c2 = cos( 2 * phi );
   x = E4 * c2 * c2 + a4 - E4;
   if ( x >= 0 )
   {
      x = sqrt( x );
      if ( E2 * c2 + x >= 0 )
      {
         r = sqrt( E2 * c2 + x );
         c = cos( phi ), s = sin( phi );
         P2d(  r * c,  r * s ).Mark( Red, Thickness );
         P2d( -r * c, -r * s ).Mark( Red, Thickness );

         r = E2 * c2 - x;
         if ( r >= 0 )
         {
            r = sqrt( r );
            P2d(  r * c,  r * s ).Mark( Red, Thickness );
            P2d( -r * c, -r * s ).Mark( Red, Thickness );
         }
      }
   }
}
```

We start by defining some local constants for frequently used terms. Next, we define local variables used in the following computation part. In a for-loop we mark the real solution points of $s(\varphi)$ in a straightforward computational way.

If there exist four real solutions on one straight line $s(\varphi_0)$, the curve consists of two parts. Then there exist two real tangents t_1 and t_2 to o through the origin. On each tangent t_i we find two points of tangency T_i and T_i' (Figure 2.15). In the regions around these points our drawing method yields a bad distribution of points. Thus, we mark them and two "neighboring" tangent points explicitly in order to get a better graphical result:

Listing from `"cassini.cpp"`:

```
const Real eps = 0.03 * a;
if ( a < E )
{
   x = 0.5 * asin( a * a / E2 );
   r = pow( E4 - a4, 0.25 ) - eps;
   for ( int i = 0; i < 3; i++, r += eps )
   {
      P2d P( r * cos( x ), r * sin( x ) );
      P.Mark( Red, Thickness );
      P.Reflect( Yaxis2d );
      P.Mark( Red, Thickness );
      P.Reflect( Xaxis2d );
      P.Mark( Red, Thickness );
      P.Reflect( Yaxis2d );
      P.Mark( Red, Thickness );
   }
}
```

The real eps depends on the parameter a and ensures a good distance of the neighboring tangent points. ♢

2.2 Animations

The previous section of this book already contains a few examples of animations in Open Geometry. However, these animations were not the main content of the respective programs. This section's examples put a special emphasis on the animation part and provide many different ways of altering position, shapes, and relations of objects. You will learn to use them efficiently for the production of real-time animations with Open Geometry 2.0.

Example 2.16. Propagation of waves
An example of a 2D animation of a physical concept is realized in the sample file `"wavefront.cpp"`. There, we consider a wave source E, (i.e., a point emitting sound waves or optical waves). According to the laws of acoustics or optics, the wave will spread to all directions with constant velocity. The set of all points on the wave belonging to the same time parameter t is called a *wave front*. If the wave remains undisturbed, all wave fronts are circles with common center E.

Suppose now that c is a reflecting curve. Each point of a wave front that meets c is reflected and continues its travel in a new direction with the same constant velocity. Depending on the shape of c and the position of E relative to c, the wave front's shape will be changed considerably. It is not difficult to prove the following theorem (compare Figure 2.16 and [30]):

After being reflected on a curve c, the wave fronts are offset curves of the orthonomic o of c with respect to E.[6]

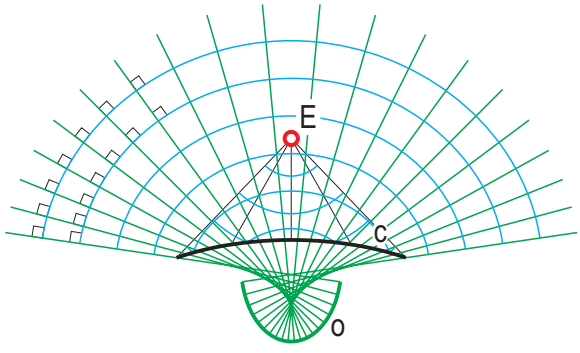

FIGURE 2.16. After the reflection on c, the wave fronts are all offset curves of the orthonomic o of c with respect to E.

In `"wavefront.cpp"` the reader can find an animation of the physical model. The reflecting curve $c \ldots C(\mathsf{u})$ is a class of type *ParamCurve2d*, each wave front to be displayed is a class of type *L2d*. We first define a function returning the point $P(\mathsf{u}, \mathsf{t})$ of the wave front (u is the parameter of the corresponding point on c, t is the time parameter). Thus, each wave front may be regarded as a u-line of a curved parameterization of the plane.

Listing from `"wavefront.cpp"`:

```
// Function returning a reflected point. t is the time parameter,
// u is the parameter value of the corresponding point on the
// reflecting curve, (i.e, the u-lines are exactly the wave fronts)
P2d WaveFront( Real t, Real u )
{
    P2d P, Q;
    Q = refl_curve.CurvePoint( u );
    V2d dir = Q - E;
    dir.Normalize( );
    P = E + t * dir;
    if ( t > E.Distance( Q ) )
        P.Reflect( refl_curve.Tangent( u ) );
    return P;
}
```

[6]The terms offset curve and orthonomic are explained on pages 47 and 44, respectively.

Now to the animation. We introduce a few parameters (global constants) to determine:

- The center E of the wave fronts.
- The parameter interval [U1, U2] of the reflecting curve (it is necessary to define U1 and U2 globally, since we must have access to them in the Draw( ) part of our file).
- The time interval [T1, T2] during which the wave fronts will be displayed. Usually we will have T1 = 0; i.e., the wave fronts will be drawn as soon as they are emitted.
- The number M of points on each wave front. Since the wave fronts tend to get rather long as t approaches the end of the time interval, you may have to adjust this in order to get "smooth" curves.
- The number N of wave fronts to be displayed.
- The velocity of the animation (SpeedFactor).

Next, we define the three variables:

Listing from `"wavefront.cpp"`:

```
const Real Step = ( T2 - T1 ) / N;  // time interval between
                                    // two wave fronts
Real delta;                         // the parameter used for the animation
L2d U[N];                           // the N wave fronts to be drawn
```

The time interval is divided into equal parts of length Step each, the real Delta is used in Animate( ), the i-th wave front itself is an *L2d*-object named U[i]. The Draw( ) and Animate( ) part of our scene read as follows:

Listing from `"wavefront.cpp"`:

```
void Scene::Draw( )
{
   ShowAxes2d( Green, -10, 10, -7, 7 );
   // define and draw N wave fronts with M points each
   int i = 0, j = 0;
   for ( i = 0; i < N; i++ )
   {
      U[i].Def( Blue, M );
      for ( j = 0; j <= M; j++ )
```

```
        {
            // to delay the wave fronts at the
            // beginning of the animation
            if ( FrameNum( ) < i * SpeedFactor )
                U[i][j] = E;
            else
            {
                Real u0 = U1 + j * ( U2 - U1 ) / ( M - 1 );
                Real t0 = T1 + i * ( T2 - T1 ) / N;
                U[i][j] =
                    WaveFront( t0 + Step * Delta / SpeedFactor, u0 );
            }
        }
        // only draw the relevant wave fronts
        if ( FrameNum( ) > i * SpeedFactor )
            U[i].Draw( THIN );
    }
    refl_curve.Draw( THICK );
    E.Mark( Red, 0.2, 0.1 );
}

void Scene::Animate( )
{
    Delta = FrameNum( ) % SpeedFactor;
}
```

We divide the "parameter range" of the wave front area into equally distributed stripes of width $(T2 - T1)/2$. In each stripe we draw one wave front per frame. Repeating this in cycles of SpeedFactor frames (compare Animate( )), we produce the illusion of wave fronts traveling with constant velocity (while in reality, they move a short distance and jump back again). Note the additional delay of the i-th wave front during the first frames of the animation.

In `"wavefront.cpp"` the reflecting curve c is an ellipse, and the wave source E is one of its focal points. Due to the focal property, the reflected wave fronts are parts of circles with center F (the other focal point of c; see Figure 2.17). ♢

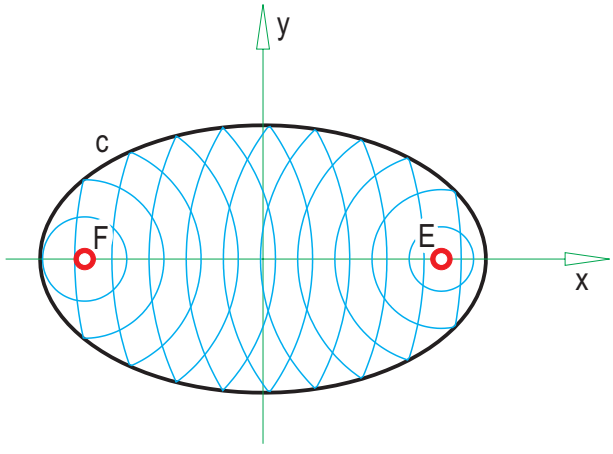

FIGURE 2.17. Output of the program `"wavefront.cpp"`.

Example 2.17. Rolling snail
The next example concerns one of the most remarkable plane curves: the *equiangular* or *logarithmic spiral*. In polar coordinates its equation reads

$$\sigma \dots r(\varphi) = \varrho e^{p\varphi} \qquad \varphi \in (-\infty, \infty).$$

The real constants ϱ and p are called the *radius* and *parameter* of the curve. The spiral has a number of beautiful properties. For example, the evolute or the catacaustic of s is an equiangular spiral again. For our purposes, the following will be of importance to us: *The angle ψ between the radius vector and the corresponding curve tangent is constant* (Figure 2.18).

We can use this to construct a stair upon which a logarithmic spiral can roll up and down (right-hand side of Figure 2.18). Our "wheel" consists of the spiral arc between the parameter values $\varphi_0 = 0$ and $\varphi_1 = 2\pi$. The length of the stairs equals the arc length of s in this interval. The stairs themselves are not orthogonally: we need a little corner for the spiral to fit in. The angles are determined by the angle ψ formed by the radius vector and curve tangent.

The corresponding Open Geometry program is called `"rolling_snail.cpp"`. In order to draw the spiral, we have to derive a class from *ParamCurve2d*. We do this and add some additional — rather special — methods that will be described in the following.

Listing from `"rolling_snail.cpp"`:

```
class EquiangularSpiral: public ParamCurve2d
{
```

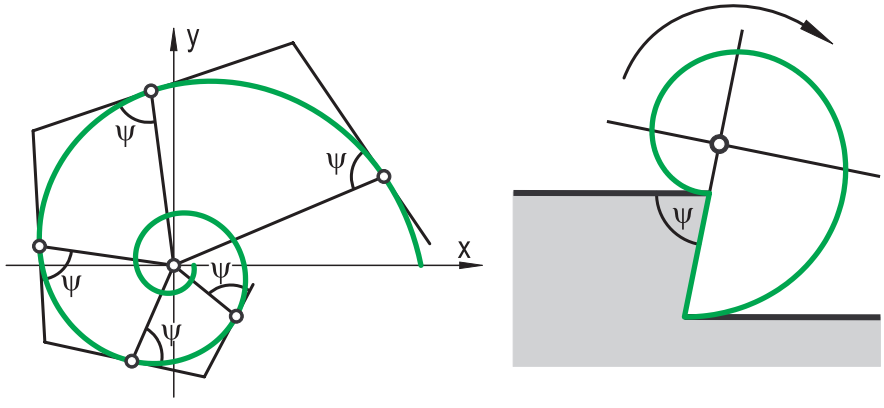

FIGURE 2.18. The radius vectors and the tangents of an equiangular spiral intersect in a constant angle ψ (left). The spiral can roll steadily on a stair (right).

```
public:
  void Def( Color c, int n, Real radius, Real parameter,
            Real t0, Real t1 )
  {
    rad = radius;
    par = parameter;
    S.Def( rad, 0 );
    E.Def( exp( 2 * par * PI ) * rad, 0 );
    ParamCurve2d::Def( c, n, t0, t1 );
  }
  virtual P2d CurvePoint( Real t )
  {
    P2d X( cos( t ), sin( t ) );
    X *= rad * exp( par * t );
    return X;
  }
  Real ParamValue( Real t0, Real dist )
  {
    return ( 2 * log( rad * sqrt( par * par + 1 ) *
           exp( par * t0 ) + dist * par ) -
           log( ( par * par + 1 ) * rad * rad ) ) / ( 2 * par );
  }
  void RotateAndDraw( ThinOrThick thickness, Real dist,
       P2d V, V2d v )
  {
    Real tt = ParamValue( 0, dist );
    P2d T1 = CurvePoint( tt );
    StrL2d normal;
    normal.Def( P2d( 0.5 * ( T1.x + V.x ), 0.5 * ( T1.y + V.y ) ),
                V2d( V.y - T1.y, T1.x - V.x ) );
```

```
        Real angle = v.Angle( TangentVector( tt ), false, true );
        Real d = 0.5 * T1.Distance( V );
        Real dd = d / tan( 0.5 * angle );
        angle = Deg( angle );
        P2d C = normal.InBetweenPoint( dd );

        Rotate( C, -angle );
        S.Rotate( C, -angle );
        E.Rotate( C, -angle );
        Origin2d.Rotate( C, -angle );

        Draw( thickness );
        StraightLine2d( col, S, E, thickness );
        Origin2d.Mark( Red, 0.2, 0.1 );

        Rotate( C, angle );
        S.Rotate( C, angle );
        E.Rotate( C, angle );
        Origin2d.Rotate( C, angle );
    }
  private:
    Real rad;
    Real par;
    P2d S;
    P2d E;
};
EquiangularSpiral Spiral;
```

We define the spiral by its parameter and radius. In addition, we set two points S and E: the start and end point of our arc. The parametric representation in rectangular coordinates reads

$$\vec{x}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \varrho e^{pt} \begin{pmatrix} \cos t \\ \sin t \end{pmatrix}$$

and is implemented in CurvePoint(...). The function ParamValue(...) has two real input parameters t1 and dist. The return parameter t1 is the explicit solution of the integral equation

$$\int_{t0}^{t1} \sqrt{\dot{x}(t) + \dot{y}(t)} = \text{dist}.$$

Geometrically speaking, we can say that the arc length between $\vec{x}$(t0) and $\vec{x}$(t1) is dist. We need this, of course, to simulate the rolling of the spiral.

For reasons of simplicity, we assume at first that σ rolls on one straight line s only. We assume further that in a starting position the start point S is the point of contact (the velocity pole of the motion; Figure 2.19).

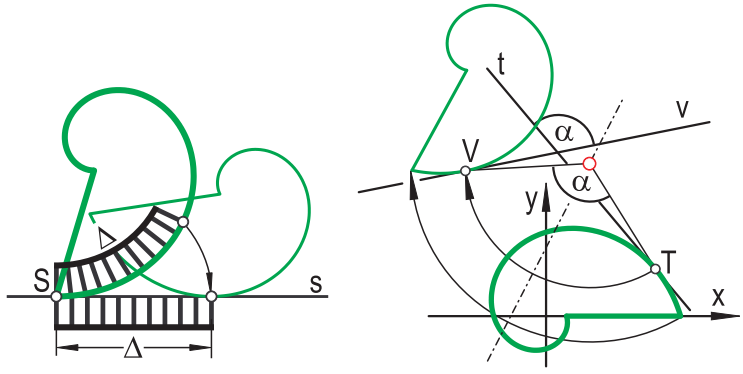

FIGURE 2.19. Rotating the spiral in the correct position.

In a neighboring position the point of contact belongs to some parameter value t (computed by ParamValue(...)) that depends on the distance Δ measured on s. We have to rotate the spiral to its correct position.

This (and a little more) is done in the rather lengthy function RotateAndDraw(...). We refer to a fixed spiral, compute the center and angle of the rotation, rotate and draw all relevant elements. Then we immediately undo the rotation. This procedure is a little complicated but necessary: The CurvePoint(...) and Tangent(...) functions of *ParamCurve2d* refer to the parameterized equation and thus return only the points and tangents of the original position. The rotation is now defined by two pairs (T, t) and (V, v) of point plus tangent (right-hand side of Figure 2.19).

Equipped with the class EquiangularSpiral, the rest of the program is easy to write. We have to adjust the length and angles of the stair with radius and parameter of the spiral, and we have to write a proper animation. We use the parameter P of the spiral and the length Length of the stair as global constants. With the help of some basic calculus we compute the radius of s as well as height and vertices of the stairs in Init( ) (N is the total number of stairs):

Listing from `"rolling_snail.cpp"`:

```
Real r = fabs( Length * P / Sqrt( 1 + P * P ) /
         ( exp( 2 * PI * P ) - 1 ) );
Real angle = atan( P );
Real h  = r * fabs( exp( 2 * P * PI ) - 1 );
Real height = h * cos( angle );

int i;
Real x = 0.5 * h * sin( angle );
```

```
for ( i = 0; i < N2; i += 2 )
    Vertex[i].Def( 0.5 * ( i - N ) * Length + i * x,
      0.5 * ( N - i ) * height );
for ( i = 1; i < N2; i += 2 )
    Vertex[i].Def( 0.5 * ( i - N + 1 ) * Length +
      ( i - 1 ) * x, 0.5 * ( N - i + 1 ) * height );
```

So far, so good. But we still have to animate the whole thing. We use three global variables for this: T, Delta, and Index. The variable T gives the current distance of the point of contact from the left vertex of the current stair. Delta is the increment for T and can change its sign as the snail rolls down and up again, Index is an integer value that stores the number of the current stair. Thus, in order to animate the scene, we have to write the simple line

```
Spiral.RotateAndDraw( THICK, T, Vertex[Index] +
                      T * Xdir2d, Xdir2d );
```

in Draw( ). The animation part itself is a little trickier. Still, it is not difficult to grasp the basic ideas:

Listing from `"rolling_snail.cpp"`:

```
void Scene::Animate( )
{
  T += Delta;
  if ( T > Length )
  {
    T = 0;
    Index += 2;
    if ( Index == N2 )
    {
      Index -= 2;
      Delta *= -1;
      T = Length;
    }
  }
  if ( T < 0 )
  {
    T = Length;
    Index -= 2;
    if ( Index == -2 )
    {
      Index = 0;
      Delta *= -1;
      T = 0;
    }
  }
}
```

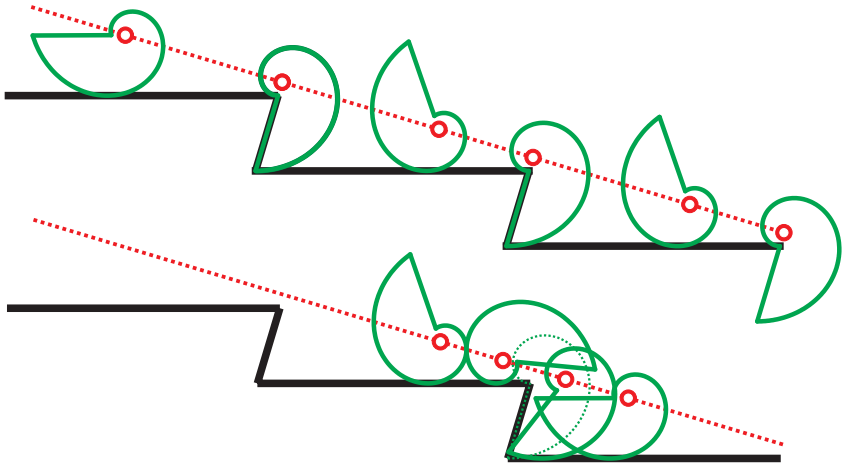

FIGURE 2.20. A snail rolls on a stair.

We increase T until we reach the end of a stair. There we reset it and increase Index by 2 (because each stair consists of two end points, and Index refers to the index of the first vertex). If we reach the end of the stair, we do everything just the other way round. In Figure 2.20 we display several positions during the rolling motion. ♢

Snails are nice but sometimes a little too slow (if there is no stair to roll on). In the next examples we will deal with faster animals.

Example 2.18. About cats, dogs, and rabbits
Suppose a dog D is chasing two cats. If it is not too smart, it probably tries to catch both at the same time. As a result, the cats will always be able to escape. In this example we are interested in finding the path curve of the hunting dog.

In `"cats_and_dog1.cpp"` we assume that the cats C_1 and C_2 do nothing but run on circular paths. We initialize a starting position of D, C_1, and C_2, the centers M1 and M2 of the cat's circular paths, and their angular velocities VelCat1 and VelCat1, respectively, by random numbers. In Draw( ) we just mark the points and draw their path curves. The really interesting parts are happening in Animate( ) (Cat1 $= C_1$, Cat2 $= C_2$ and Dog $= D$ are globally declared variables of type *P2d*):

Listing from `"cats_and_dog1.cpp"`:

```
void Scene::Animate( )
{
  V2d dog_dir = Cat1 + Cat2;
  dog_dir -= 2 * Dog;
```

```
    if ( dog_dir.Length( ) > VelDog )
    {
        dog_dir.Normalize( );
        dog_dir *= VelDog;
    }
    Dog.Translate( dog_dir );
    DogRect.Translate( dog_dir );

    Cat1.Rotate( M1, VelCat1 );
    Cat1Rect.Translate( Cat1.x - Cat1Rect[1].x,
        Cat1.y - Cat1Rect[1].y );
    Cat2.Rotate( M2, VelCat2 );
    Cat2Rect.Translate( Cat2.x - Cat2Rect[1].x,
        Cat2.y - Cat2Rect[1].y );
}
```

At the beginning we compute the instantaneous velocity vector of the dog D. Its direction is given by $\overrightarrow{DC_1} + \overrightarrow{DC_2} = \overrightarrow{OC_1} + \overrightarrow{OC_2} - 2\overrightarrow{OD}$. If it is not too short, we normalize it and multiply it by the absolute value of the dog's velocity. The rectangles attached to the points D, C_1, and C_2 are just there to illustrate the theme of the program. They will be textured with dog and cat bitmaps.

If you run the program you will notice that sooner or later, the dog always follows a certain curve that seems to be independent of the dog's starting position and resembles a trochoid of characteristic VelCat1 : VelCat2 (Figure 2.21).

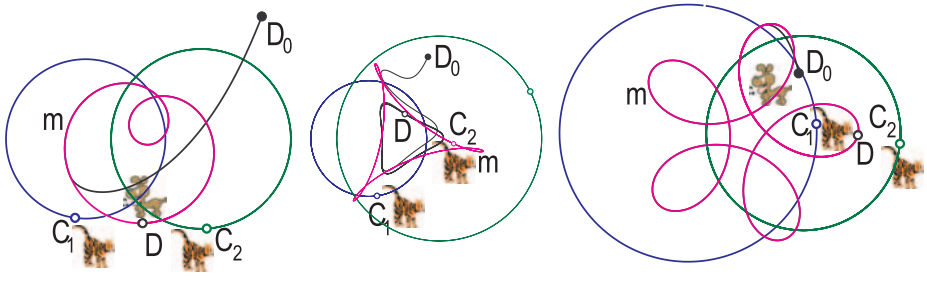

FIGURE 2.21. Some examples of the dog's path curve. The initial position of the dog is D_0.

A closer examination reveals the underlying geometric structure. To begin with, it is clear that the dog at any moment tries to reach the midpoint M of C_1 and C_2 (Figure 2.22). If it is fast enough, it can reach it, and it then tries to stay there. Because of this behavior, we will refer to the path curve m of M as the *attractor curve*. Even if C_1 and C_2 have constant velocity, M will usually not have. Thus, if M accelerates too much, the dog has to start hunting its imaginary target again. Have a look at the middle image of Figure 2.21, where the dog's velocity was chosen relatively low.

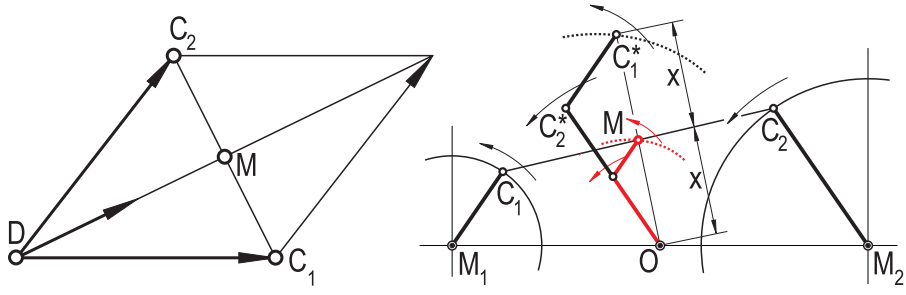

FIGURE 2.22. The dog D tries to reach the midpoint M of C_1 and C_2. The path curve of M is a trochoid.

It is not difficult to see that the attractor is a trochoid (Figure 2.22). Just translate the rotating rod M_2C_2 through the midpoint O of M_1M_2 and M_1C_1 through C_2^*. The midpoint of O and C_1^* is at the same time the midpoint of C_1 and C_2. Thus, the path curve of M is really a trochoid.[7] Its characteristic $\omega_1 : \omega_2$ is given by the angular velocities of C_1 and C_2. Its dimensions are half the radii of the path circles of C_1 and C_2.

In `"cats_and_dog2.cpp"` we display the kinematic generation of the path curve of M. Additionally, we delay the drawing of the dog's path curve in order to emphasize its generation.

A variation of the program is `"rabbit_and_dogs.cpp"`. There, two wild dogs are hunting a poor rabbit. Again, the dogs D_1 and D_2 are not too clever and head directly for the rabbit R. The rabbit's direction is determined by the wish to flee from the dogs and to reach a sheltering cave C. The directions are computed in Animate( ):

Listing from `"rabbit_and_dogs.cpp"`:

```
void Scene::Animate( )
{
  if ( !Hit && !Save )
  {
    V2d dog1_dir = Rabbit - Dog1;
    V2d dog2_dir = Rabbit - Dog2;
    V2d rabbit_dir = Cave - Rabbit;
```

[7]The dog's path is, of course, not a trochoid in general. It is for example, possible that D and M are identical only for a while. This shows that the dog's path is not analytic in this case and therefore cannot be a trochoid.

```
        dog1_dir.Normalize( );
        dog2_dir.Normalize( );
        rabbit_dir.Normalize( );

        rabbit_dir = dog1_dir + dog2_dir + 2 * rabbit_dir;
        rabbit_dir.Normalize( );
        rabbit_dir *= VelRabbit;
        Rabbit.Translate( rabbit_dir );
        RabbitRect.Translate( rabbit_dir );
        dog1_dir *= VelDog1;
        Dog1.Translate( dog1_dir );
        Dog1Rect.Translate( dog1_dir );
        dog2_dir *= VelDog2;
        Dog2.Translate( dog2_dir );
        Dog2Rect.Translate( dog2_dir );
    }
}
```

Hit and Safe are global variables of type *Boolean*. They are initialized with false and can change their value at the end of Draw( ):

Listing from `"rabbit_and_dogs.cpp"`:

```
if ( Dog1.Distance( Rabbit ) < DoubleLimit ||
     Dog1.Distance( Rabbit ) < DoubleLimit )
{
    V2d v( Double( ), Double( ) );
    Rabbit.Translate( v );
    RabbitRect.Translate( v );
}
if ( Dog1.Distance( Rabbit ) < HitLimit ||
     Dog2.Distance( Rabbit ) < HitLimit )
    Hit = true;
if ( Rabbit.Distance( Cave ) < SafetyLimit )
    Save = true;
```

If the rabbit is close enough to the cave, it is safe. If it comes too close to a dog, it is lost and Hit is set to true. As the situation is really unfair (*two* dogs that run *faster* than the poor rabbit), we introduce the possibility of doubling. The rabbit makes a sudden jump in a random direction (Double( ) yields a suitable random number). Thus, the rabbit will get caught only if it doubles in a bad random direction. In this case, the animation stops and we enter the following path in Draw( ):

Listing from `"rabbit_and_dogs.cpp"`:

```
if ( Hit )
{
   Rabbit.AttachString( Red, 0.5, 0.5, "HIT" );
   if ( FrameNum( ) % 20 < 10 )
   {
      Dog1.Mark( Black, 0.2, 0.1 );
      Dog2.Mark( Black, 0.2, 0.1 );
      Rabbit.Mark( Red, 0.4 );
   }
   else
   {
      Rabbit.Mark( Blue, 0.2, 0.1 );
      Dog1.Mark( Black, 0.2, 0.1 );
      Dog2.Mark( Black, 0.2, 0.1 );
   }
}
```

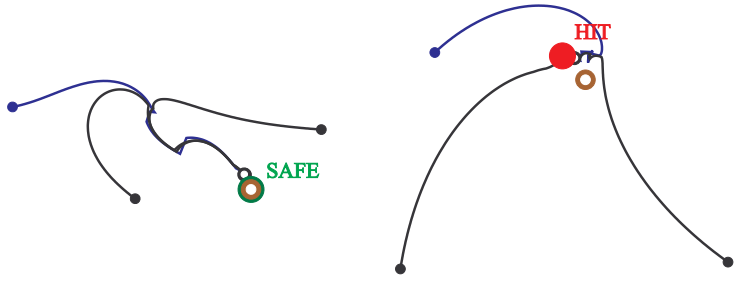

FIGURE 2.23. Two different endings of the rabbit hunt.

This will highlight the rabbit's position with a blinking red point and attach the string `"HIT"` to it. An analogous path exists if the rabbit reaches the cave. Figure 2.23 shows two typical endings of the hunt: In the image on the left-hand side the rabbit escapes after doubling twice. In the image on the right-hand side it is caught near the sheltering cave. ♢

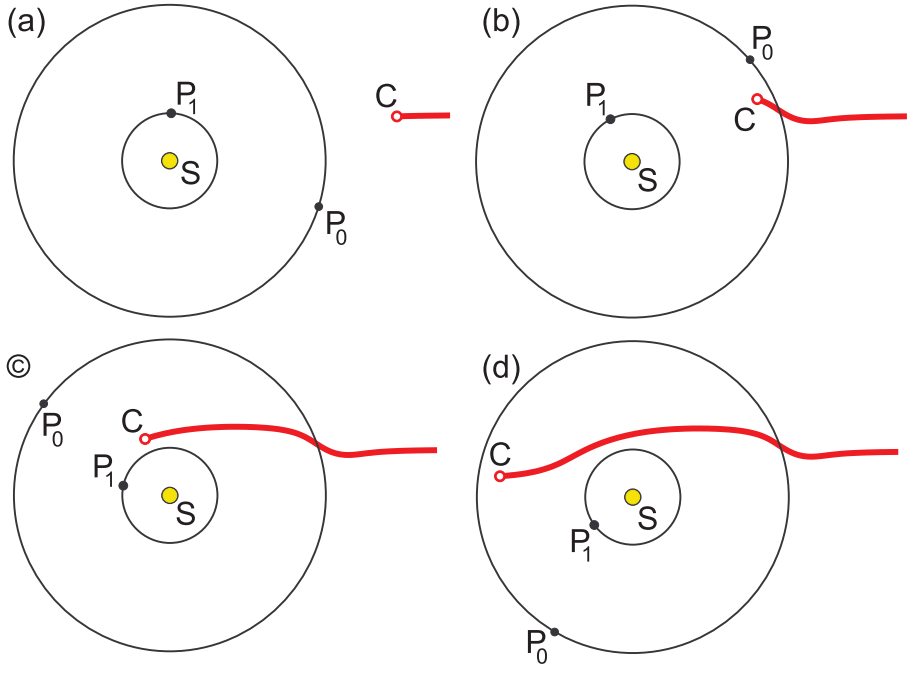

FIGURE 2.24. (a) The "lucky" comet heads toward planet P_0 and the system consisting of S and P_1. (b) C is turned aside by P_0 but does not crash into the planet. (c) All three forces of gravity guide the comet through the solar system. (d) C safely leaves the solar system.

Example 2.19. A comet passes a solar system
A variation of the theme of the cat and rabbit hunt is to be found in `"comets_fate.cpp"`: A comet enters a system of planets that orbit around a sun. Initially, the planet's velocity is constant, but under the influence of the sun's and planets' forces of gravity, it will change its path. Eventually, it will either be able to leave the solar system or crash into one of the planets or the sun. Similar simulations may be applied not only to large objects such as planets but also to small objects like interacting electrons or a crowd of people heading in different directions and trying to avoid too close contact.

In `"comets_fate.cpp"` we decide on a global constant integer N that gives the number of planets in the solar system. Each planet Planet[i] has its own force of gravity g_i, its circular velocity ω_i, and its path circle c_i. The center of c_i is the fixed sun S_i. According to Newton's law, the gravity that acts on the comet C is proportional to the planet's mass and inversely proportional to the square of the distance $\overline{CP}$. The comet has a certain starting point and a certain initial velocity $\vec{v}_0$. The starting point is on the far right of the screen, and it is no loss of generality to assume $\vec{v}_0$ as parallel to the x-axis.

In Init( ) we initialize the variables with sensible random values; in Draw( ) we add the comet's current position to a 2D path curve, draw the path curve and the circular paths and mark the planets (in a size proportional to their force of gravity), the sun and the comet.[8] The part of interest is Animate( ):

Listing from `"comets_fate.cpp"`:

```
void Scene::Animate( )
{
   int i;
   if ( !Hit )
   {
      // Rotate the planets.
      for ( i = 0; i < N; i++ )
         Planet[i].Rotate( Sun, Omega[i] );

      // Compute the distances of comet
      // to planets and the sun.
      Real Rad, R[N];
      Rad = Comet.Distance( Sun );
      for ( i = 0; i < N; i++ )
         R[i] = Comet.Distance( Planet[i] );

      // If comet is too close to sun or other planets
      // we consider it as lost in the atmosphere.
      if ( GravSun > Rad * Rad * Rad )
      {
         Comet = Sun;
         Hit = true;
      }
      for ( i = 0; i < N; i++ )
      {
         if ( Grav[i] > R[i] * R[i] * R[i] )
         {
            Comet = Planet[i];
            Hit  = true;
         }
      }
      // If comet is not too close to sun or other
      // planet it moves under the influence of
      // gravity. The force of gravity is reciprocal
      // to the square of the distance to the planet.
      if ( !Hit )
      {
```

[8] Of course, the resulting system of planets cannot be stable! We use random values and do not describe the situation in a real solar system.

```
            V2d v = Sun − Comet;
            v.Normalize( );
            Comet += GravSun / Rad / Rad ∗ v;
            for ( i = 0; i < N; i++ )
            {
               v = Planet[i] − Comet;
               v.Normalize( );
               Comet += Grav[i] / R[i] / R[i] ∗ v;
            }
            Comet += CVel;
         }
      }
      else
      {
         // If comet hits planet we mark it with
         // a blinking red point.
         if ( FrameNum( ) % 20 < 10 )
            Comet.Mark( Red, 0.4 );
      }
   }
```

We use a global variable Hit of type *Boolean* that tells us whether the comet has crashed into a planet or the sun. If Hit = **false**, we orbit the planets to their new position. Then we compute the distances Rad and R[i] of the comet to the sun and these planets. This is necessary because the force of gravity acting on C is proportional to the reciprocal value of the squared distance.

We will translate the comet C by a vector of length $g_i/\mathsf{R}[\mathsf{i}]^2$ in the direction of the planet P_i (and in the direction of the sun as well). But first we have to decide whether C has already hit a planet. This part needs some consideration. Obviously, the crash distance δ_i (i.e., the distance when we assume that the comet hit the planet) must depend on the force of gravity. A sensible limit is $\delta_i = \sqrt[3]{g_i}$. Otherwise, our model would catapult the comet through the planet P_i if it is within the circle of radius δ_i around P_i. In the listing we have used the equivalent expression $g_i > \mathsf{R}[\mathsf{i}]^3$.

If the comet is too close to a planet, we set Hit = **true** and $C = P_i$. Otherwise, we apply the translations described above. Of course, we must take into account the initial velocity $\vec{v}_0$ (CVel in our listing) as well. If Hit is **true**, we will not change the planet's position in the next frame. Instead, we mark the crash position with a gleaming red point. The comet's chances to survive its travel through the solar system increase with its initial velocity and decrease with the number of planets. Figure 2.24 shows the typical fate of a comet. ♢

2.3 Kinematics

Open Geometry serves very well for computer simulations of rigid 2D mechanisms. In [14] a whole chapter is dedicated to this topic. In addition to this, will give a few more examples.

Example 2.20. A complex window opener
At first, we present a sample program that is used to develop a manipulator for performing a complex opening and closing procedure of a window. The manipulator has to meet the following few conditions:

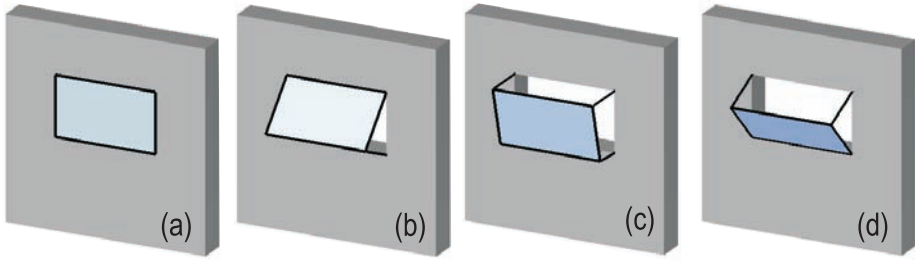

FIGURE 2.25. Four different positions of the window: (a) completely closed, (b) half open and protecting from the sun, (c) completely open, (d) half open and reflecting the sun rays inside the room.

- One must be able to reach the four window positions displayed in Figure 2.25. These positions guarantee optimal usage of the sunlight. A half-transparent silvered surface either reflects the light inside the room or prevents it from coming through in the half-open positions.
- On the one hand, a transition from position (b) to position (c) should be possible without closing the window completely, i.e., without reaching position (a). On the other hand, one should be able to close the window directly from position (b) or (d).[9]
- For practical reasons the mechanism should be simple and stable at the same time. Only standard gearing devices should be used. The essential joints should all fit into the window frame.

It is a difficult task to find a mechanism of that kind, and Open Geometry cannot do it for you. You rather need a good book on manipulators and a certain amount of experience. At a certain point, however, an Open Geometry simulation might be quite helpful.

[9]These are essential elements of design. Without them the mechanism would be of much less interest from the designer's point of view.

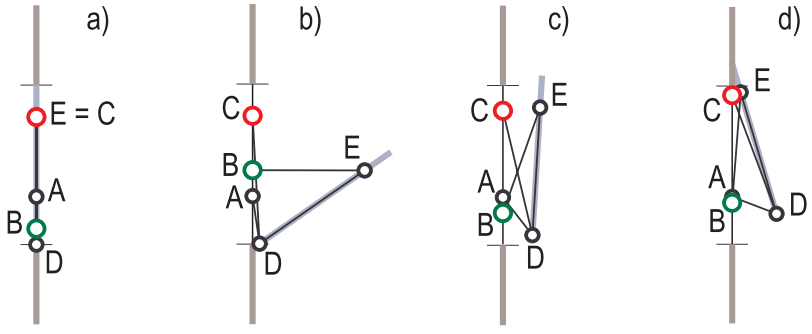

FIGURE 2.26. The abstract mechanism of the window manipulator.

Obviously, the problem can be solved in two dimensions only. There, you need a manipulator providing a two-parameter mobility of freedom in order to meet the second condition of the above list. The final proposal is the mechanism displayed in Figure 2.26.

It consists of four arms AD, BE, CD, and DE that are all linked by joints of revolution. The points The points The points The points The points The points The points The points The points A, B, and C always lie on a straight line w (the wall); A is fixed, B and C may still change their position on w. This gives us the two parameters of freedom we need.

Having decided on the abstract mechanism, another difficult task is ahead: the design of the mechanism's dimensions. You must be able to reach the different positions without crashing the window into the wall and without sweeping off the flowers from your window sill. And what if one of the manipulator's arms is not long enough to reach one of the required positions? In earlier days you had to make a large number of drawings and build a few models to find a good solution. Nowadays, you write an OPEN GEOMETRY program (`"window.cpp"`)!

What has to be done? Given the dimensions $\overline{\mathsf{AD}}$, $\overline{\mathsf{CD}}$, $\overline{\mathsf{BE}}$, and $\overline{\mathsf{DE}}$ of the mechanism and the positions of A, B, and C, we have to compute the positions of D and E. We do this in two almost identical functions called CalcD(...) and CalcE(...). Only one of them needs to be listed.

Listing from `"window.cpp"`:

```
P2d CalcD( P2d inA, P2d inC )
{
    // Two circles with centers A and C. Their radii
    // are AD and CD, respectively.
    Circ2d a, c;
    a.Def( NoColor, inA.x, inA.y, AD );
    c.Def( NoColor, inC.x, inC.y, CD );
```

```
        // Intersect these circles.
        P2d D1, D2;
        n = a.SectionWithCircle( c, D1, D2 );
        if ( n == 0 ) // no real solution
        {
            ResetPoints( );
            return D_old;
        }
        else
        {
            // Take the point that lies closer to the old position of D.
            // This will probably be the right choice.
            if ( D1.Distance( D_old ) < D2.Distance( D_old ) )
                return D1;
            else
                return D2;
        }
    }
```

The two arguments of CalcD(...) are the points A and C. We define two circles a and c around these points with radii $\overline{AD}$ and $\overline{CD}$, respectively. Obviously, D lies on both circles a and c. So we use the method SectionWithCircle(...) of *Circ2d* to determine the intersection points D1 and D2 of a and c. Here, the first problems arise: What happens if D1 and D2 are not real, and which point shall we take instead?

SectionWithCircle(...) returns an integer value n telling us the number of real solutions. If there is no real solution (n = 0), we return to the previous stage of our animation by resetting all points and returning the previous position D_old of D (this will soon be explained in detail). This corresponds perfectly to the situation in practice: The mechanism simply won't move in the desired direction. If two solutions exist, we take the one that lies closer to the previous position D_old of D. This may yield a wrong solution in some strange constellations, but for practical purposes it is absolutely sufficient.

It is now easy to draw the whole scene since all relevant points are known. The Animate( ) part is, however, a bit tricky. Nevertheless, it has simple code:

Listing from "window.cpp":

```
void Scene::Animate( )
{
   B_old = B;
   C_old = C;
   D_old = D;
   E_old = E;
   B.Translate( 0, Delta_B );
   Delta_B = 0;
   C.Translate( 0, Delta_C );
   Delta_C = 0;
   D = CalcD( A, C );
   E = CalcE( B, D );
}
```

We store the current positions of B, C, D, and E, translate B and C by a certain amount along w, and compute the new positions of D and E. The reason for the immediate resetting of Delta_B and Delta_C to 0 will soon become clear.

We want to change the window position *interactively*. In Open Geometry this can be done as follows

1. Write an Animate( ) part that in general, does *nothing* relevant. In our example, the value of Delta_B and Delta_C is 0 in most frames.
2. In Draw( ) or Animate( ) ask for the last key that has been pressed and change some animation variables according to it.
3. Immediately reset these animation variables in Animate( ).

In order to animate the scene interactively, you start a series of identical frames by pressing <Ctrl + f> or clicking on the corresponding button. Then you press one of your specified command keys, and the next frame will yield the desired changes. The relevant part of "window.cpp" is as follows:

Listing from "window.cpp":

```
int key = TheKeyPressed();
switch (key )
{
case 'k':
   Delta_B = Factor_B * 0.1;
   break;
case 'j':
```

```
    Delta_B = −Factor_B ∗ 0.1;
    break;
  case 'd':
    Delta_C = Factor_C ∗ 0.1;
    break;
  case 'f':
    Delta_C = −Factor_C ∗ 0.1;
    break;
  case 'i':
    Factor_B ∗= 2;
    break;
  case 'u':
    Factor_B ∗= 0.5;
    break;
  case 'e':
    Factor_C ∗= 2;
    break;
  case 'r':
    Factor_C ∗= 0.5;
    break;
  }
```

The effects of the different command keys are as follows: Pressing **k** moves the point B up by a certain increment; pressing **j** moves it down again. The key **i** doubles the increment; **u** halves it again. Thus, the velocity of B will be increased or decreased, accordingly.[10] Analogous commands are provided for the point C.

Just two more hints:

- Use PrintString(...) to display the meaning of your command keys on the screen. It will help you and any other user to handle the program correctly.
- Some keys or key combinations are occupied for other purposes, so you can't use them. You will find a list of all occupied keys in the "Help" menu of the Open Geometry window.

♢

[10]This turned out to be necessary, since a constant increment has its drawbacks. If it is too large, you cannot reach certain positions of the mechanism because you jump from a good position (two solutions to both D and E) immediately to a bad position (no solution to either D or E). If the increment is too small, the animation is too slow to be fun.

Example 2.21. Mechanical perspective
Our next example is the 2D animation of a *perspectograph*. This is a mechanical device to produce perspective figures of a given plane figure F. Its kinematics are not too complicated (compare Figure 2.27):

- A point M runs along the figure F.
- Two lines l_1 and l_2 connect M with two fixed points O_1 and O_2.
- l_1 and l_2 intersect a guiding line s parallel to O_1O_2 in two points P_1 and P_2.
- P_1 and P_2 are connected to two points R_1 and R_2 on s.
- R_1 and R_2 are two opposite vertices of a rhombus r_1. The edges of the rhombus are connected through joints of revolution.
- R_1 is the edge of congruent rhombus r_2, so that the edges of r_1 and r_2 intersect orthogonally at R_1.
- With r_2 we connect a point N by doubling an edge of r_2 that does not contain R_1.
- The described mechanism provides a two-parameter mobility, and N is always located on a figure F' perspective to F.

We have described the mechanism in a way that is already suitable for an OPEN GEOMETRY program (`"perspectograph.cpp"`). There, we start by defining some global variables that — together with the position of M — determine the initial position.

Listing from `"perspectograph.cpp"`:

```
const P2d O1( 9, 3 );
const P2d O2( O1.x, -7 );

const Real Dist = 3.5;
const Real Length = 0.9 * Dist;

const StrL2d S( Origin, V2d( O2, O1 ) );
```

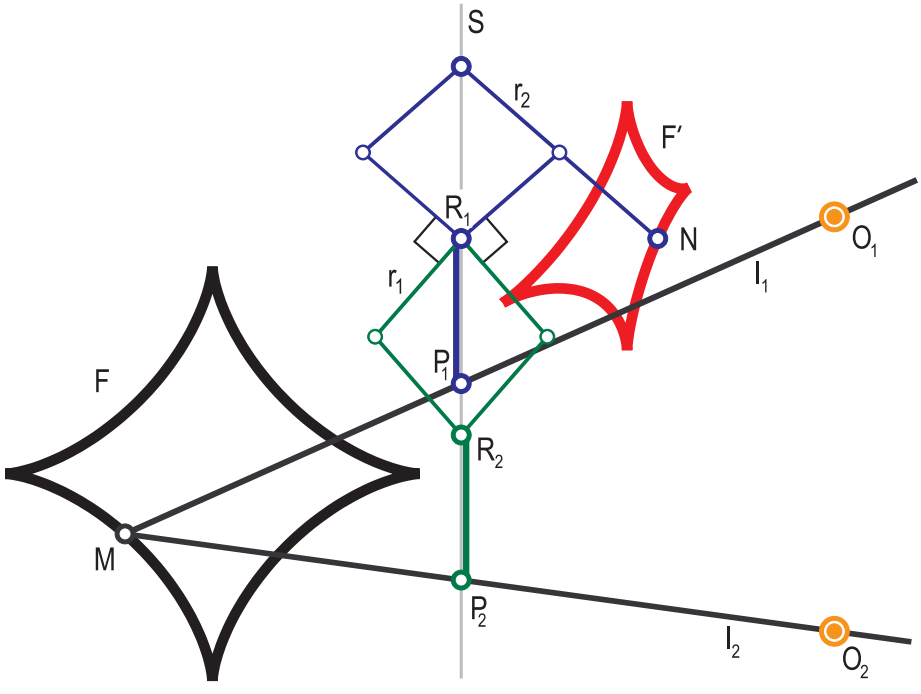

FIGURE 2.27. The perspectograph.

Dist is the distance between the points P_1 and R_1 (or P_2 and R_2); Length is the side length of the rhombi; and S is the guiding line parallel to O_1O_2. In the next step we write a function called ComputeAndMark(...). Its return value is void; its arguments are a 2D point M and a variable add_point of type *Boolean*. It computes and marks all relevant points and lines for the current position of M. If add_point is true, it adds M and N to two path curves PathM and PathN, respectively.

We could do everything directly in Draw( ) but — since the code is rather lengthy — we preferred separating it from Draw( ). It is not worth listing the contents of ComputeAndMark(...) here. There is just one important thing: After computing P_1 and P_2 we check whether their distance is larger than $2 \cdot$ Length. It is easy to see that $\overline{P_1P_2} =$ Length. Thus, $\overline{P_1P_2} > 2 \cdot$ Length yields an invalid position of the mechanism. In this case, we give a warning and do not draw anything.

Now we can define a path for M. In our example we take the parameterized equation of an astroid.

Listing from `"perspectograph.cpp"`:

P2d PathPoint(*Real* t)

```
{
    Real r = 5;
    return P2d( -r * pow( cos( t ), 3 ) - 6,
                r * pow( sin( t ), 3 ) - 3.2 );
}
```

Draw( ) and Animate( ) are simply as follows:

Listing from "perspectograph.cpp":

```
void Scene::Draw( )
{
    P2d X = PathPoint( T );
    if ( X.Dist( S ) < 0.3 )
        Delta *= -1;
    ComputeAndMark( X, true );
}
void Scene::Animate( )
{
    T += Delta;
}
```

Here, T is a globally declared real. We take the path point and check whether it is too close to the guiding line S. If so, we change the orientation of the path curve by taking the negative increment. This is desirable for two reasons:

1. In practice, M cannot pass the guiding line due to mechanical reasons.
2. In the computer model there might occur discontinuities in the computation of the relevant points (The mechanism "jumps" from one position to the next).

By now, everything should work fine. The output of the program is displayed in Figure 2.27. ♢

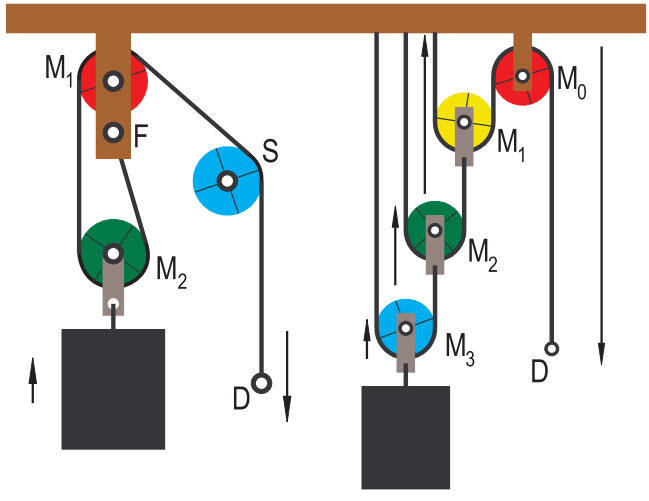

FIGURE 2.28. A simple pulley block (left) and a sophisticated one (right). If N is the number of pulleys, the necessary pulling force is just $1/N$ of the weight.

Example 2.22. Pulley blocks
A pulley block is a well-known mechanical device that allows the lifting of heavy loads with rather small effort. In fact, there exist different kinds of pulley blocks. In `"pulley_block1.cpp"` we present the simplest case (Figure 2.28).

A rope is fixed at a point F and runs over three pulleys with centers M2, M1, and S. The points M1 and S are fixed, while M2 is suspended on the rope. On this last pulley a heavy load is attached. Pulling down the end point D of the rope by a vector $\vec{v}$ will lift the load by $\frac{1}{2}\vec{v}$. Since the work on both ends has to be the same, the load seems to have only half of its actual weight.

Now to the animation in `"pulley_block1.cpp"`. We define F, M2, S, and the pulley radius Rad as global constants. Since the rest of the drawing is really not time-critical, we will do it completely in Draw( ). We use a pulsing real L for the animation. It gives the length M1M2. We initialize it as follows:

Listing from `"pulley_block1.cpp"`:

```
L.Def( 0.5 * ( Min + Max ), -0.05, Min, Max, HARMONIC );
```

Here Min = 3 * Rad and Max = Min + 4.5. This ensures that the pulleys will never intersect during the animation. The pulleys themselves are objects of type *RegPoly2d*, which gives us the possibility of high-quality output on the screen by

deciding on a relatively high number of vertices.[11] The drawing of the pulleys, the suspensions, and the weight are not really exciting. Let's have a look at the drawing of the rope instead. It consists of line and arc segments. Quite often we have to determine the tangents of a circle through a given point. We use a function to do this task:

Listing from `"pulley_block1.cpp"`:

```
int CircleTangents( P2d Mid, Real Rad, const P2d &P, P2d T[2] )
{
   Real d = Mid.Distance( P );
   if ( d < fabs( Rad ) )
      return 0;
   if ( d == Rad )
   {
      T[0] = P;
      T[1] = P;
      return 1;
   }
   Real x = Rad * Rad / d;
   Real y = Sqrt( Rad * Rad - x * x );
   V2d v = P - Mid;
   v.Normalize( );
   T[0] = Mid + x * v + y * V2d( -v.y, v.x );
   T[1] = Mid + x * v - y * V2d( -v.y, v.x );
   return 2;
}
```

The input parameters are the circle's center and radius, the given point, and two more points for the points of tangency to be stored. The return value gives the number of real solutions. With the help of CircleTangents(...) and the Open Geometry class *Sector2d*, it is not difficult to draw all rope segments:

[11] The Open Geometry class *Circ2d* automatically adjusts the number of points on the circumference of the circle. That is, small circles are regular polygons with relatively few vertices.

Listing from `"pulley_block1.cpp"`:

```
StrL2d s;
P2d D( S.x + Rad, S.y - 2 * Max - 2 + 2 * l );
s = Yaxis2d.GetParallel( -D.x );
s.Draw( Black, D.y, S.y, THICK );

s.Def( S, M1 );
s = s.GetParallel( -Rad );
s.Draw( Black, 0, S.Distance( M1 ), THICK );

Sector2d arc;
arc.Def( Black, P2d( S.x + Rad, S.y ), S,
        s.InBetweenPoint( 0 ), 10, EMPTY );
arc.Draw( true, THICK );
...
```

The remaining segments are implemented similarly. If you just draw the rope, the pulleys, and the suspension parts, the animation is not too impressive. Because of this, we draw little line segments on the pulleys that give a better sense of the rotation. The code for the first pulley reads as follows:

Listing from `"pulley_block1.cpp"`:

```
V2d dir( cos( l / Rad ), sin( l / Rad ) );
StrL2d s1( M1, dir );
StrL2d s2 = s1;
s2.Rotate( M1, 90 );
s1.Draw( Black, -Rad, Rad, THIN );
s2.Draw( Black, -Rad, Rad, THIN );
```

Here, l is the current value of the pulsing real L. We transform it to the arc length on the pulley by dividing it by the radius. Then we define and draw two straight lines through M1 that correspond to this length. We do the same for the pulleys around M2 and S. The pulley around M2 rotates, however, with only half-angular velocity, i.e., the transformation to the arc length reads $l \mapsto l/2r$.

A slightly more sophisticated pulley block is animated in `"pulley_block2.cpp"` (Figure 2.28). There, a series of N pulleys is used. The system of any two consecutive pulleys can be seen as a pulley block of the first kind. Thus, the necessary pulling force is reduced to $1/2^N \cdot$ weight. Of course, the weight will rise only very slowly if N is too big.

What are the essential programming differences? First, we need more pulleys, which are initialized in Init( ):

Listing from "pulley_block2.cpp":

```
M[0].Def( 0.5 * ( N + 1 ) * Rad, 6.5 );
M[1].Def( M[0].x - 2 * Rad, M[0].y - DeltaY );
int i;
for ( i = 2; i < N; i++ )
{
    M[i] = M[i-1];
    M[i].Translate( -Rad, -DeltaY );
}
```

In Draw( ) we compute their new position:

Listing from "pulley_block2.cpp":

```
Real l = L.Next( );
int i;
for ( i = 1; i < N; i++ )
    M[i].y = M[0].y - i * DeltaY + l / pow( 2, i - 1 );
```

Rad and DeltaY are global constants; L is a pulsing real varying harmonically between 0 and DeltaY; M[0] is the center of the fixed pulley. The pulley center M[i] approaches the height of M[0] with constant velocity $v/2^i$. The visualization of the pulley rotation is similar to the previous example:

Listing from "pulley_block2.cpp":

```
RegPoly2d Pulley;
StrL2d s;
V2d v;
Real phi;
for ( i = 0; i < N; i++ )
{
    Pulley.Def( Col[i%7], M[i], Rad, 30, FILLED );
    Pulley.Shade( );
    phi = l / Rad / pow( 2, i - 1 );
    if ( i == 0 )
        phi *= -1;
    v.Def( cos( phi ), sin( phi ) );
    s.Def( M[i], v );
    s.Draw( Black, -Rad, Rad, THIN );
    s.Rotate( M[i], 90 );
    s.Draw( Black, -Rad, Rad, THIN );
}
```

We shade the pulley with a color from a certain pool of seven colors. We compute the angular velocity (taking into account that the fixed pulley has an inverse sense of rotation) and draw straight lines to evoke a sense of rotation.

Since there are only half arc segments this time, we do not use the class *Sector2d* but an object of type *L2d* for the drawing of the rope segments. We define it in Init( ) and translate it to the appropriate position in Draw( ):

Listing from "pulley_block2.cpp":

```
HalfCircle.Def( Black, 20 );
Real t, delta = PI / 19;
for ( i = 1, t = 0; i <= 20; i++, t += delta )
    HalfCircle[i]( Rad * cos( t ), -Rad * sin( t ) );

...

for ( i = 1; i < N; i++ )
{
    HalfCircle.Translate( M[i].x - HalfCircle[1].x + Rad,
                          M[i].y - HalfCircle[1].y );
    HalfCircle.Draw( THICK );
    ...
}
```

The semicircle of the fixed pulley needs special treatment:

Listing from "pulley_block2.cpp":

```
HalfCircle.Reflect( Xaxis2d );
HalfCircle.Translate( M[0].x - HalfCircle[1].x + Rad,
                      M[0].y - HalfCircle[1].y );
HalfCircle.Draw( THICK );
HalfCircle.Reflect( Xaxis2d );
```

The drawing of the straight line segments, the suspensions, and the weight is not too difficult, so we do not display it here. ♢

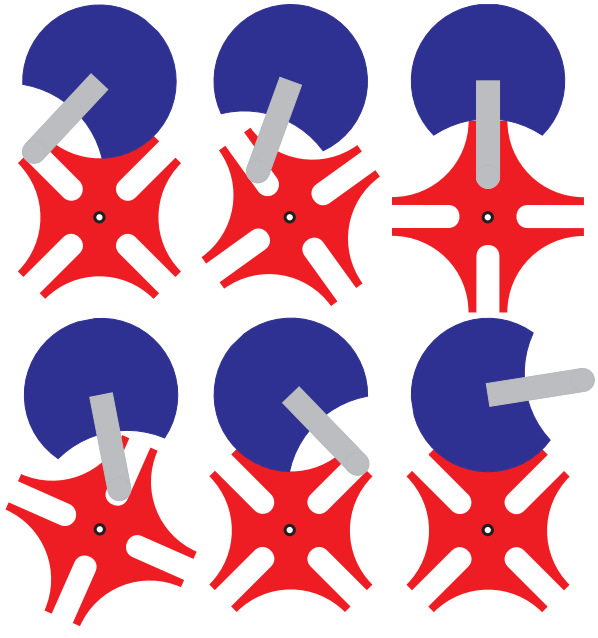

FIGURE 2.29. A mechanism with periods of standstill ("Maltesien gear").

Example 2.23. Maltesien gear
Sometimes it is necessary to use a mechanism with periods of standstill. An example of this is the "Maltesien gear" (Figure 2.29). There, a crank rotates with constant angular velocity. It interacts with a figure of special shape resembling a Maltesien cross. Transmission occurs only during one quarter of a rotation period. During the remaining period the Maltesien cross stands still.

Before implementing this mechanism in Open Geometry, we need to explore its geometry. Take a look at the third figure in the top row of Figure 2.29 as base figure. The basic shape of the Maltesien cross and the crank are a square of side length $2A$ and a circle of radius r_1. From them we subtract certain circular and rectangular regions. The dimensions can, however, not be chosen arbitrarily:

The crank and the cross touch each other during three quarters of the motion. This gives the distance $\sqrt{2}A$ of their centers and the radius r_1 of the circular subtractions at the corners of the cross (Figure 2.30). Now consider the middle position in the second row of Figure 2.29 and the auxiliary drawing in Figure 2.30. There, the beginning of the standstill periods is displayed. The dimensions of the triangle OPC and the law of cosines give a relation between r_2 and a resulting in $r_2 = A(\sqrt{2}-1)$. Finally, the circular subtraction of the crank disk is centered at O and has radius $\sqrt{A^2+b^2}$, where $b = A - r_1$.

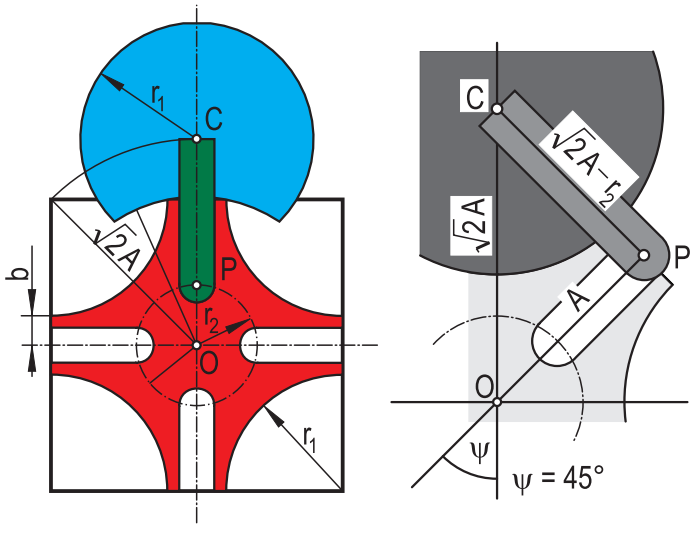

FIGURE 2.30. The mechanism's dimensions.

The implementation of the mechanism was realized in `"maltesian_gear.cpp"`. The shape of crank and cross are quite unusual, and their design needs some consideration. We begin with the crank. It consists of a circular disk of which another circle is subtracted, a rectangle, and a small circle. The circular disk will be an object of type *ComplexPoly2d*.[12] We make some definitions in Init( ):

Listing from `"maltesian_gear.cpp"`:

```
Poly2d p[1];
const int m1 = 7, m2 = 20;
const int n = 2 * ( m1 + m2 - 1 );
p[0].Def( NoColor, n, FILLED );

Real rad = sqrt( A * A + b * b );
Real omega = 0.5 * PI - atan( ( A + b ) / ( A - b ) );

Real phi, delta = omega / ( m1 - 1 );
for ( i = 1, phi = 0.5 * PI; i <= m1; i++, phi -= delta )
    p[0][i]( rad * cos( phi ), rad * sin( phi ) );

delta = 0.75 * PI / m2;
for ( i = m1+1, phi = -0.25 * PI + delta;
```

[12] We have to use a complex polygon in order to display the disk's non convex outline correctly.

```
                                    i <= m1 + m2; i++, phi += delta )
    p[0][i]( r1 * cos( phi ), r1 * sin( phi ) + sqrt( 2 ) * A );

for ( i = 2; i < m1 + m2; i++ )
    p[0][n-i+1]( -p[0][i].x, p[0][i].y );
p[0][n] = p[0][1];
Crank1.Def( Blue, 1, p );
```

In the first line we introduce an auxiliary "array" p[1] of one ordinary *Poly2d* object. In the last line we define our complex polygon. Its loops consist of the elements of p (i.e., there exists only one loop, as is indicated by the second parameter of the defining method). The computation of the points is a little tricky, and we must pay attention not to get confused with the indices.

It would be even harder to define the cross in the same way. But there is an alternative to computing the points' coordinates: We simply draw a rectangle in red and cover it with a few circles and rectangles filled in pure white.

Listing from `"maltesian_gear.cpp"`:

```
MaltRect1.Def( Red, 2 * A, 2 * A, FILLED );
MaltRect1.Translate( -A, -A );

int i;
for ( i = 0; i < 4; i++ )
    MaltRect2[i].Def( PureWhite, A - r2, 2 * r3, FILLED );
MaltRect2[0].Translate( -A, -r3 );
MaltRect2[1].Translate( r2, -r3 );
MaltRect2[1].Rotate( Origin, 90 );
MaltRect2[2].Translate( r2, -r3 );
MaltRect2[3].Translate( r2, -r3 );
MaltRect2[3].Rotate( Origin, -90 );

for ( i = 0; i < 4; i++ )
    InvMalt1[i].Def( PureWhite, Origin, r1, FILLED );
InvMalt1[0].Translate( -A, -A );
InvMalt1[1].Translate( -A,  A );
InvMalt1[2].Translate(  A, -A );
InvMalt1[3].Translate(  A,  A );
for ( i = 0; i < 4; i++ )
    InvMalt3[i].Def( PureWhite, Origin, r3, FILLED );
InvMalt3[0].Translate( -r2,  0 );
InvMalt3[1].Translate(  r2,  0 );
InvMalt3[2].Translate(  0, -r2 );
InvMalt3[3].Translate(  0,  r2 );
```

The two remaining parts CrankBar and CrankEnd are a simple rectangle and circle, respectively. The important thing in Draw( ) is now the drawing order of the objects in order to get the correct visibility:

Listing from `"maltesian_gear.cpp"`:

```
void Scene::Draw( )
{
   MaltRect1.Shade( );
   int i;
   for ( i = 0; i < 4; i++ )
      MaltRect2[i].Shade( );
   for ( i = 0; i < 4; i++ )
      InvMalt1[i].Draw( THIN );
   for ( i = 0; i < 4; i++ )
      InvMalt3[i].Draw( THIN );

   Crank1.Shade( );
   CrankBar.Shade( );
   CrankEnd.Draw( THIN );
}
```

In Animate( ) we rotate the crank parts through a certain angle increment about their common center. Then we have to determine the corresponding rotation of the cross. We use a global real variable Psi_old for that.

Listing from `"maltesian_gear.cpp"`:

```
void Scene::Animate( )
{
   Crank1.Rotate( Center, Delta );
   CrankEnd.Rotate( Center, Delta );
   CrankBar.Rotate( Center, Delta );

   Real psi = Deg( atan( CrankEnd.Mid( ).x / CrankEnd.Mid( ).y ) );

   int i;
   if( Psi_old - psi <= 0 )
   {
      MaltRect1.Rotate( Origin, Psi_old - psi );
      for ( i = 0; i < 4; i++ )
      {
         MaltRect2[i].Rotate( Origin, Psi_old - psi );
         InvMalt1[i].Rotate( Origin, Psi_old - psi );
         InvMalt3[i].Rotate( Origin, Psi_old - psi );
      }
   Psi_old = psi;
}
```

We compute the new angle denoted by ψ in Figure 2.30. (In a general position it is, of course, not equal to 45°.) The previous angle is already stored in Psi_old. We rotate the cross and the invisible circles and rectangles by Psi_old − psi, but only if the sense of rotation is negative. Hence, a rotation occurs only for $\psi \in [-45°, 45°]$, and that is exactly what we want. ♢

Now to something different. We will still deal with planar kinematics but from a more theoretical point of view.

Example 2.24. The kinametic map
A motion in the Euclidean plane $\mathbb{R}^2$ is a transformation $\mu : \mathbb{R}^2 \to \mathbb{R}^2$ that leaves both the distance of any two points and the orientation of any triangle unchanged. In a Euclidean coordinate system it reads

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = M \begin{pmatrix} x \\ y \end{pmatrix} + \vec{t}$$

where M is an orthogonal 2×2 matrix of determinant 1, and $\vec{t}$ is a translation vector. In general, μ is a rotation about a certain center C. If C is a point at infinity, the transformation is a pure translation. The manifold of all Euclidean motions depends on three parameters (the coordinates of the center of rotation and the rotation angle). Hence, its cardinality is equal to the cardinality of $\mathbb{R}^3$. In other words; it should be possible to map the points of $\mathbb{R}^3$ bijectively to the motions of $\mathbb{R}^2$ in a natural way.

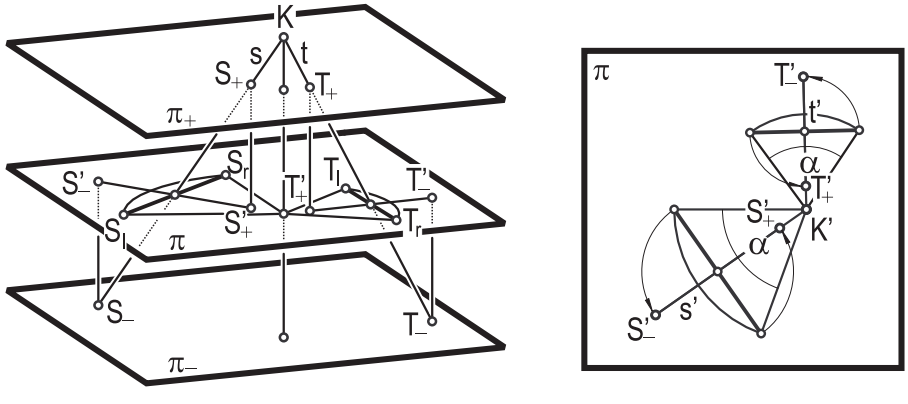

FIGURE 2.31. The kinematic map.

The first map of that kind was described in works by E. Study, W. Blaschke, and J. Grünwald and is called a *kinematic map*. We will denote it by κ. The visualization of κ combines some of Open Geometry's favored tasks: 2D kinematics, 3D curves, and geometric constructions in space. Hence, we will have a closer look at it:

The kinematic map κ has a very lucid geometric interpretation. Let π be a plane in $\mathbb{R}^3$. We use a Euclidean coordinate system with $x, y \subset \pi$ and identify $\mathbb{R}^2$ with π. By Σ we denote the set of straight lines in $\mathbb{R}^3$ that are not parallel to π; by Π we denote the set of points of π. We choose a real $D \neq 0$ and introduce two planes $\pi_+ \ldots z = D$ and $\pi_- \ldots z = -D$. Then we can define a bijective map

$$g\colon \Gamma \to \Pi \times \Pi; \qquad s \mapsto g(s) = \{S_\mathrm{l}, S_\mathrm{r}\} \tag{4}$$

in the following way (compare Figure 2.31):

1. Intersect the straight line s with π_+ and π_- in order to get two points S_+ and S_-.
2. Let S'_+ and S'_- be the normal projections of S_+ and S_- into π.
3. Rotate S'_+ and S'_- about their midpoint M through an angle of 90°. The resulting points are S_r and S_l.

Then, S_r and S_l are called *right* and *left* image points of s. In Figure 2.31 we display the right and left image points of two straight lines s and t, respectively. The straight lines s and t have a point K in common. For this reason, there exists a rotation about the normal projection K' of K on π that brings S_l to S_r and T_l to T_r. Reversed, pairs $\{S_\mathrm{l}, S_\mathrm{r}\}$ and $\{T_\mathrm{l}, T_\mathrm{r}\}$ of points corresponding in a rotation always stem from intersecting lines. If we admit points at infinity, this is not only true for rotations but for translations as well. Hence, we have a one-to-one correspondence between the motions of $\mathbb{R}^2$ and the points of $\mathbb{R}^3$: the kinematic map κ.

An Open Geometry implementation is to be found in `"kinematic_map1.cpp"`. There exist formulas to compute the kinematic counter image $K = \kappa^{-1}(\mu)$ of a motion μ when the center K' and the rotation angle α of μ are given. It is, however, more convenient to construct K from two pairs of corresponding points. We use the following function for that task:

Listing from `"kinematic_map1.cpp"`:

```
void SetPoints( const P3d Pl, const P3d Pr,
     P3d &P_plus, P3d &P_minus )
{
   P3d M = 0.5 * ( Pl + Pr );
   StrL3d axis( M, Zdir );
   P_plus = Pl;
   P_minus = Pr;
   P_plus.Rotate( axis, 90 );
   P_minus.Rotate( axis, 90 );
   P_plus.z = D;
   P_minus.z = -D;
}
```

SetPoints(...) sets the values of P_+ and P_- when the pair $\{P_l, P_r\}$ of image points is given, D is a global variable to determine the auxiliary planes $\pi_+, \pi_- \ldots z = \pm$D.

Now we need a one-parameter set of planar motions. Here, we can refer to OPEN GEOMETRY's large stock of motion classes. In our program we use the motion induced by a four-bar linkage ("coupler motion"; see [14]). Since everything takes place in $[x, y]$, it is not even necessary to adapt the class *FourBarLinkage* for use in 3D. We define an instance Mechanism of this class and define it in Init( ) according to the examples given in [14].

The four-bar linkage consists of four points L, M, A, and B, where L and M are fixed points, A rotates about L, B rotates about M, A and B are linked by a rod of constant length. We denote the initial positions of A and B by A_l and B_l, respectively. Every new position of the mechanism defines new points A_r, B_r and a motion μ with $A_\mathrm{l} \mapsto A_\mathrm{r}$, $B_\mathrm{l} \mapsto B_\mathrm{r}$. By means of the inverse kinematic map κ^{-1}, we will get a curve $k \subset \mathbb{R}^3$ that should be displayed. We store the values of A_l and B_l in global variables and construct the points of k in Draw( ):

Listing from `"kinematic_map1.cpp"`:

```
Ar.x = Mechanism.GetA( ).x;
Ar.y = Mechanism.GetA( ).y;
Br.x = Mechanism.GetB( ).x;
Br.y = Mechanism.GetB( ).y;
SetPoints( Al, Ar, A_plus, A_minus );
StrL3d a( A_plus, A_minus );
StrL3d b( B_plus, B_minus );
P3d K = a * b;
KinPath.AddPoint( K );
KinPath.Draw( THICK, STD_OFFSET, 10 );
```

The variables A_r and B_r are global of type *P3d*; their z-coordinate, however, is always set to zero. KinPath is an instance of the class *PathCurve3d*. In order to get an attractive picture, we mark important points (including A_+, A_-, B_+, and B_-) and draw some connecting lines. In addition, we shade three rectangular frames that represent $[x, y]$ and the auxiliary planes π_+ and π_-. We want to be able to watch everything in $[x, y]$. So it seems sensible to use opacity. For this reason, the shading has to be the last thing done in Draw( ).

Figure 2.32 shows the output of the program. It can be proved that if S_l remains fixed while S_r varies on a circle, the straight line $\kappa^{-1}(S_\mathrm{l}, S_\mathrm{r})$ moves on a regulus. In our example, this is the case for two pairs $\{A_\mathrm{l}, A_\mathrm{r}\}$, $\{B_\mathrm{l}, B_\mathrm{r}\}$ of points. The kinematic image k is therefore the intersection of two quadrics and hence a space curve of order four. Special dimensions of the mechanism may, however, cause a splitting of k into two components.

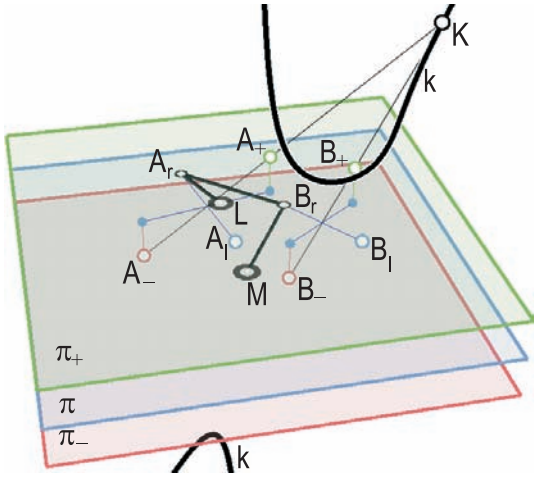

FIGURE 2.32. The kinematic counter image of a coupler motion is a space curve k of order four.

We can, of course, proceed in the reverse order as well (`"kinematic_map2.cpp"`). A space curve k defines a one-parameter set of motions in $[x, y]$. We can use the kinematic map κ to draw the trajectory of a point P_0 or to construct the polodes of these motions.

In addition to the SetPoints(...) method of the previous example, we need the inverse method (i.e., the map g of equation (4)) as well. Given a straight line s, we want to set the left and right image points P_l and P_r:

Listing from `"kinematic_map2.cpp"`:

```
void InvSetPoints( const StrL3d s, P3d &Pl, P3d &Pr )
{
    Plane pi_plus, pi_minus;
    pi_plus.Def( P3d( 0, 0, D ), Zdir );
    pi_minus.Def( P3d( 0, 0, -D ), Zdir );
    Pl = pi_plus * s;
    Pr = pi_minus * s;
    Pl.z = 0;
    Pr.z = 0;
    StrL3d axis( 0.5 * ( Pl + Pr ), Zdir );
    Pl.Rotate( axis, -90 );
    Pr.Rotate( axis, -90 );
}
```

We introduce the counter image k of the motion through a parameterized equation and call it KinCIm (kinematic counter image). In order to transform a point by the rotation μ that corresponds to a point $K(k_x, k_y, k_z)^T \in k$, we use a little formula and the following function:

Listing from `"kinematic_map2.cpp"`:

```
P3d TransformPoint( const P3d Pl, Real u )
{
   P3d K = KinCIm.CurvePoint( u );
   Real omega = atan( -D / K.z );
   K.z = 0;
   P3d Pr = Pl;
   Pr.Rotate( StrL3d( K, Zdir ), 2 * Deg( omega ) );
   return Pr;
}
```

The center of μ is the normal projection of K onto π. The angle ω of the rotation is given by

$$2\omega = \arctan\Big(-\frac{D}{k_z}\Big),$$

where $\pm D$ are the z-coordinates of the planes π_+ and π_-, respectively. Now, how can we get the polodes of the motion? This is easy, since the following result holds:

The polodes of a Euclidean motion can be found by applying the map g to the tangents of the kinematic counter image k of the motion. The set of left image points is the fixed polode, the set of right image points is the moving polode.

This is easy to implement in Open Geometry. We will draw the polodes as path curves, but we could as well derive parametric representations. In Draw( ) we compute the poles F and M of fixed and moving polodes and add them to the corresponding path curves (the kinematic counter image k is implemented as a parameterized curve, and the global real variable U refers to a curve point of k):

Listing from `"kinematic_map2.cpp"`:

```
InvSetPoints( KinCIm.Tangent( U ), F, M );
FixedPolode.AddPoint( F );
MovingPolode.AddPoint( M );
FixedPolode.Draw( MEDIUM, STD_OFFSET, 10 );
MovingPolode.Draw( MEDIUM, STD_OFFSET, 10 );
F.Mark( PureRed, 0.2, 0.1 );
M.Mark( PureGreen, 0.2, 0.1 );
```

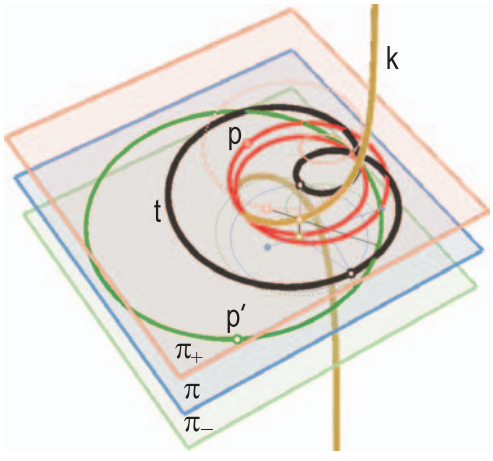

FIGURE 2.33. One trajectory t, the fixed polode p, and the moving polode p' of the kinematic image of a cubic circle k.

An example image is displayed in Figure 2.33. We additionally marked a few points and drew connecting lines in order to make the construction more lucid. ♢

2.4 Fractals

Since B. MANDELBROT published his classic *"Fractal Geometry of Nature"* in 1977 ([23]) fractals have become very popular. They play an important role in physics, chemistry, biology, astronomy, meteorology, statistics, economy, mathematics, and — last but not least — computer graphics. The definition of a fractal is, however, controversial. MANDELBROT defines it as a set of fractal dimension higher than topological dimension. Of course, he has to explain the fractal dimension first. The book [7] takes a more intuitive (and less rigorous) point of view and gives some typical properties of a fractal F:

- F is finely structured.
- F is too irregular to be described by traditional geometric means.
- F is very often self-similar or almost self-similar.
- The fractal dimension of F (whatever that is) is usually higher than its topological dimension.
- Very often F can be easily defined (e.g, by a recursive formula).[13]

[13]For this reason, fractals are usually easy to implement in a computer program!

These properties will be enough for our task in this book: the programming of a few fractal images.

Example 2.25. Koch fractals
We start with a very simple but huge class: the *Koch fractals*. According to MANDELBROT, we need two things to define a Koch fractal: an *initiator* and a *generator*. The initiator is just a starting *teragon* (that is what polygons are called in this context). The generator is a map that takes two points (i.e., a line segment) as input parameters and returns a sequence of points (a polygon). Both, initiator and generator, can usually be displayed in a simple graphic (Figure 2.34).

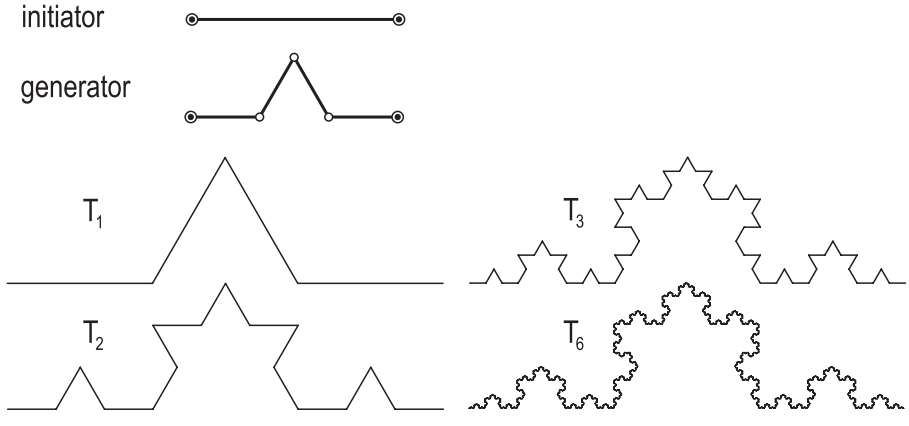

FIGURE 2.34. Same teragons of the Koch curve and its initiator and generator.

In our first example, the initiator is a straight line. The generator divides the segment into three parts of equal length and returns the vertices of the equiangular triangle over the middle part. Now we can define a sequence $\langle T_0, T_1, T_2, \dots \rangle$ of teragons by

1. applying the generator to each edge of the initiator T_0;
2. building a new teragon T_1 of the vertices of T_0 and the newly generated points;
3. repeating the same procedure with the edges of T_1.

Figure 2.34 shows the teragons T_1, T_2, T_3, and T_6. The fractal F to this initiator and generator is defined as

$$F := \lim_{i \to \infty} T_i. \tag{5}$$

In our case, F is the famous *Koch curve*. Note that T_6 is already a very good approximation for F.

OPEN GEOMETRY provides the class *KochFractal*. It is derived directly from *O2d* and allows the fast and comfortable generation of Koch fractals. Take, e.g., a look at `"koch_curve.cpp"`.

In some respects, Koch fractals in OPEN GEOMETRY are similar to parameterized curves: The class *KochFractal* is an *abstract class* due to its purely virtual member function Generator( ). This means that we have to derive our own class from KochFractal and write our own Generator( ) function:

Listing from `"koch_curve.cpp"`:

```
class MyFractal: public KochFractal
{
public:
   virtual void Generator( void )
   {
      int n = 3;

      int n1 = n + 1;
      int size1 = Size( ) - 1;
      int new_size = size1 * n + Size( );
      int i, j;

      P2d *Q = new P2d [Size( )];
      for ( i = 0; i < Size( ); i++ )
         Q[i] = GetPoint( i );
      O2d::Def( NoColor, new_size );

      for ( i = 0, j = 0; i < size1; i++, j += n1 )
      {
         pPoints[j] = Q[i];
         V2d v( Q[i], Q[i+1] );
         v /= 3;
         pPoints[j+1] = pPoints[j] + v;
         pPoints[j+3] = pPoints[j+1] + v;
         v.Rotate( 60 );
         pPoints[j+2] = pPoints[j+1] + v;
      }

      pPoints[new_size-1] = Q[size1];
      IncreaseStep( );
      delete [] Q;
   }
};
MyFractal KochCurve;
```

The integer n is the number of new points that are added to each segment of the initiator. It may vary according to the stage of development of the fractal. Later we will see examples of this. The following part (until the for loop) should not be changed by the reader. It allocates new memory for the next teragon.

The for loop is the place to write the actual generator. There the reader has to describe the new points pPoints[j+1],...,pPoints[j+n] in terms of Q[i] and Q[i+1] (be careful not to mix up the indices!). In our example we build the characteristic equiangular triangle of the Koch curve.

The generator is called in the animation part of the program. Thus, with each new frame a new teragon is created. You will notice that the generator is independent of the initial previous teragon. That is you can change the initiator without changing the generator. The initiator itself is more or less identical to OPEN GEOMETRY's Init( ) part

Listing from `"koch_curve.cpp"`:

```
void Scene::Init( )
{
   P2d P[2];
   P[0]( -10, -3 );
   P[1]( 10, -3 );
   KochCurve.Def( 2, P );
}
```

We get an array P of points and define the Koch fractal. The first integer parameter of *KochFractal*::Def(...) is the number of points in P.

Draw( ) is very simple. It reads:

Listing from `"koch_curve.cpp"`:

```
void Scene::Draw( )
{
   Silent( );
   KochCurve.Draw( Red, THIN );
}
```

The Silent( ) command is necessary in order to turn off the OPEN GEOMETRY message "warning: more than 30000 points!". Consider our simple example. The number p_n of points of T_n can be computed by the recursion

$$p_n = 4p_{n-1} - 3.$$

Thus after 8 steps we exceed the limit of 30000 points for the first time (65537 points). After a few more steps the computing time for each new frame will be too long for nervous computer programmers. So you had better not start the auto-animation![14]

Changing the initiator immediately yields different Koch curves. The following is taken from "koch_snowflake.cpp":

Listing from "koch_snowflake.cpp":

```
void Scene::Init( )
{
  const int m = 3;
  P2d P[m+1];
  P[0]( 5, 0 );
  for ( int i = 1; i <= m; i++ )
  {
    P[i] = P[0];
    P[i].Rotate( Origin, -360 / m * i );
  }
  KochCurve.Def( m + 1, P );
}
```

The initiator is a regular triangle. Note that it consists of 3 sides and therefore of 4 points. If you calculate the points via a rotation through an angle of $+360 \cdot i/m$, the orientation of the triangle changes and you will get a different fractal. Both types are displayed in Figure 2.35.

The generator may vary together with the stage of development of the fractal. In order to control this, the class *KochFractal* provides the function GetStep( ) that returns the current step. In "alt_koch_curve.cpp" we used this to build the equiangular triangles on the left- or right-hand side of the edges according to the parity of the current step. In "random_koch_fractal1.cpp" we allowed a random choice of the side for each new step, in "random_koch_fractal2.cpp" we changed it for every single edge. We print the essential part of the generator for the last case (rnd( ) is an instance of *RandNum* and produces a random number in $[-1, 1)$).

[14] If you do it by accident, it will cause no harm to kill the process.

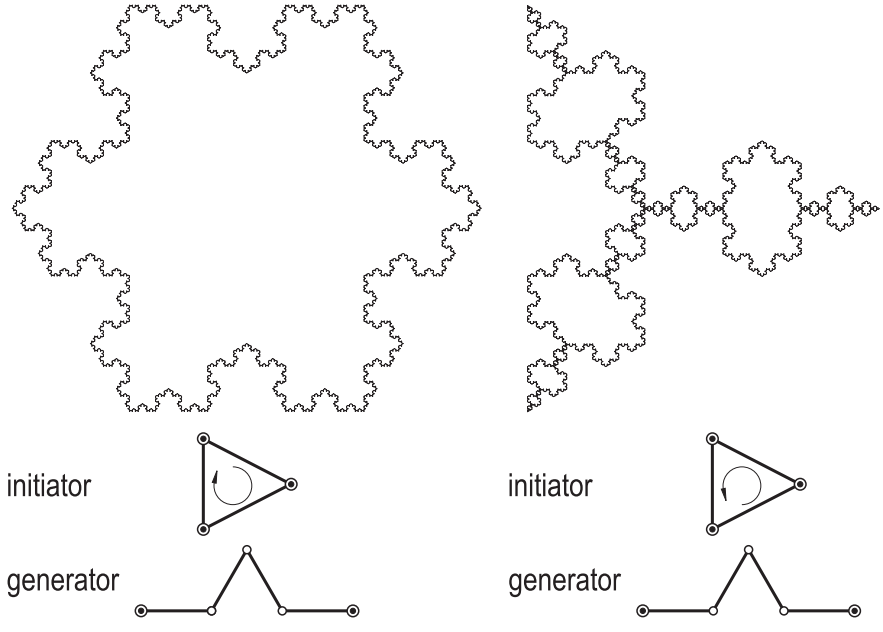

FIGURE 2.35. Two fractals of Koch type. The left fractal is called Koch's snowflake.

Listing from `"random_koch_fractal2.cpp"`:

```
int k = 1;
for ( i = 0, j = 0; i < size1; i++, j += n1 )
{
   if ( rnd( ) > 0 )
   k = -1;
   pPoints[j] = Q[i];
   V2d v( Q[i], Q[i+1] );
   v /= 3;
   pPoints[j+1] = pPoints[j] + v;
   pPoints[j+3] = pPoints[j+1] + v;
   v.Rotate( k * 60 );
   pPoints[j+2] = pPoints[j+1] + v;
}
```

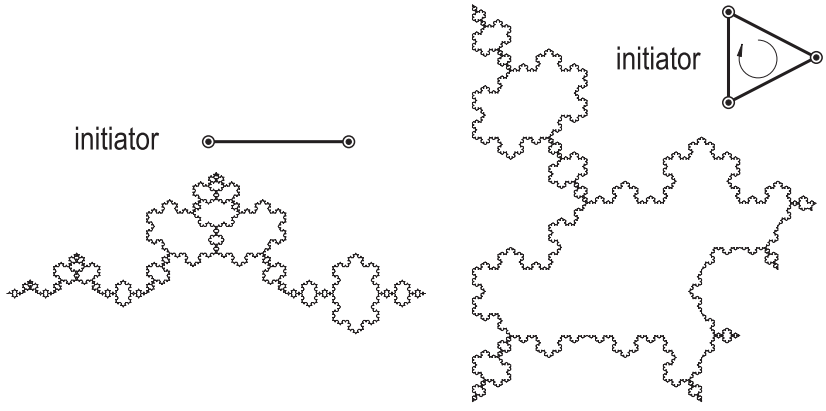

FIGURE 2.36. Two random fractals of Koch type.

Figure 2.36 shows two output examples of this program. ♢

Now it is up to you! Try different initiators and generators and explore the astonishing variety of Koch fractals. We will present just a few more examples in order to demonstrate what is possible. A real classic Koch fractal is the original PEANO curve: a curve developed by G. PEANO in 1890. He wanted to show how a curve can fill an object of two dimensions, e.g., a square (`"peano_curve.cpp"`). The initiator is a square; the generator is displayed in Figure 2.37.

Example 2.26. Autumn leaves
Another interesting example can be found in `"dragon_filling_curve.cpp"`. There, the Koch fractal fills another fractal, called "the Dragon". The initiator is a line segment; the generator is displayed in Figure 2.38. With each new segment the generator changes the side. That is, the new point lies on the right side of the first segment, on the left side of the second segment, etc.

Listing from `"dragon_filling_curve.cpp"`:

```
for ( i = 0, j = 0; i < size1; i++, j += n1 )
{
   pPoints[j] = Q[i];
   V2d v( Q[i], Q[i+1] );
   v /= 2;
   V2d w( v.y, -v.x );
   if ( i % 2 )
      w *= -1;
   pPoints[j+1] = pPoints[j] + v + w;
}
```

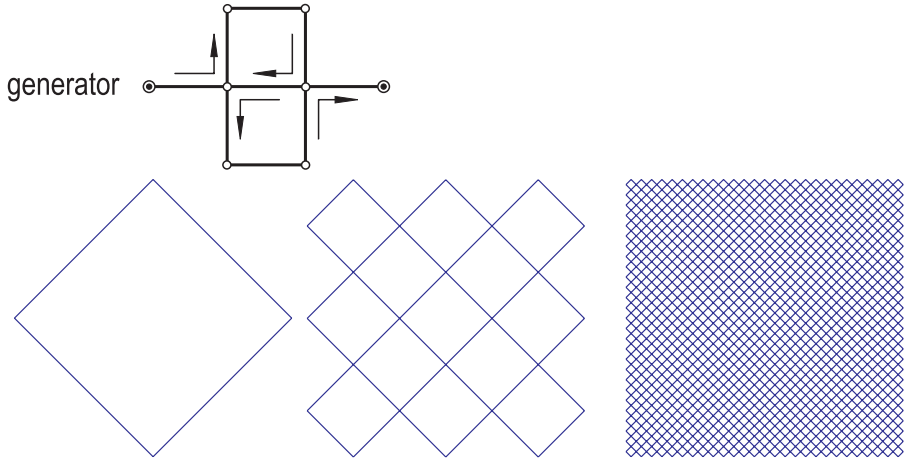

FIGURE 2.37. Steps 1, 2, and 4 in the development of the original Peano curve.

The Dragon has an interesting property: It can be used to pave the plane. Take the initiator of a Dragon-filling curve (a line segment) and rotate it three times about one of its end points through 90°. These initiators will create four Dragons that pave a certain region of the plane (Figure 2.39). If we take a square lattice of points and use each vertex four times to develop a Dragon, we can (theoretically) pave the whole plane.

In `"autumn_leaves.cpp"` we did this for a lattice of 6×4 points. The Dragons are drawn in random colors and create a pattern resembling fallen autumn leaves. Figure 2.39 is a nice picture, but you really should not miss the opportunity to watch the development of the Dragon leaves on your computer screen. ♢

Example 2.27. Newton fractals
Beside fractals of Koch type, there exist quite a few other classes of fractal sets. We present a popular example: Take an arbitrary polynomial $P(z)$ of degree n with complex coefficients. In the algebraic sense, it has n zeros $\zeta_1, \ldots, \zeta_n$.

By $N(P, z)$ we denote NEWTON's iteration sequence $\langle z_0, z_1, z_2, \ldots \rangle$ for the polynomial P with start point z, i.e., the sequence

$$z_0 := z, \qquad z_{n+1} := z - \frac{P(z_n)}{P'(z_n)}.$$

Now we define a map $\mathcal{N}\colon \mathbb{C} \to \{\zeta_1 \ldots \zeta_n, \infty\}$ by

$$z \to \begin{cases} \zeta_i & \text{if } N(P, z) \text{ converges to } \zeta_i, \\ \infty & \text{otherwise.} \end{cases}$$

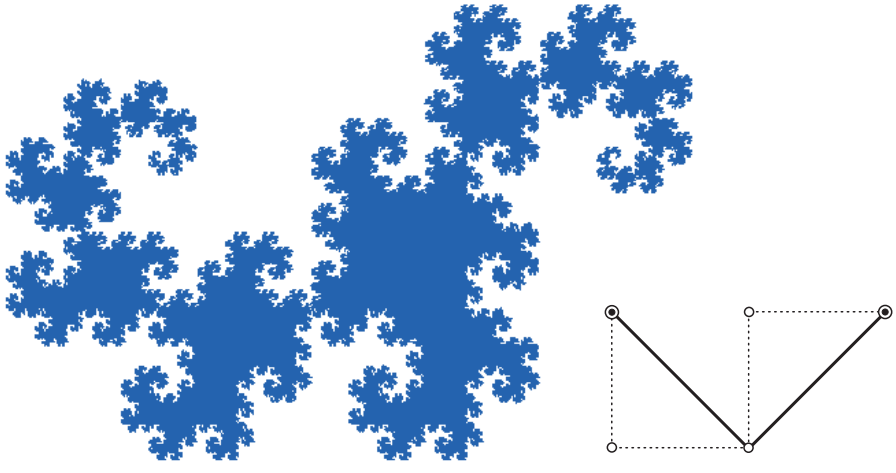

FIGURE 2.38. The first and second teragons (dotted) of the Dragon-sweeping curve and the Dragon.

Thus, $\mathcal{N}^{-1}(\zeta_i)$ is the "region" attracted to ζ_i by NEWTON's iteration. We put the word "region" in quotation marks because, in general, it is a twisted and torn subset of $\mathbb{C}$; in other words, a fractal.

The two programs we wrote for the visualization of fractals of NEWTON type are `"cubic_newton_fractal.cpp"` and `"quartic_newton_fractal.cpp"`. They are almost identical. Note that P is a cubic polynomial in the first, and a quartic polynomial in the second program. In this book we will describe the first program only.

The basic strategy is easy and straightforward: We take a cubic polynomial P and compute its zeros ζ_0, ζ_1, and ζ_2. To each zero we assign a color c_i, e.g., yellow to ζ_0, red to ζ_1, and green to ζ_2. Then we take a complex number $z = z_0$ from a fine rectangular grid and check whether it is "sufficiently close" to a zero ζ_i of P. If this is the case, we paint the corresponding pixel in the color c_i. If z_0 is not close enough to a zero of P, we apply NEWTON's iteration to z_0 and perform the same check for the new value z_1, etc. After a certain maximum number of futile attempts, we quit the loop and leave the pixel white.

In `"cubic_newton_fractal.cpp"` we take the start points z_i from the intersection of the rectangular region $[-\mathsf{X0}, \mathsf{X_0}]\times[-\mathsf{Y_0}, \mathsf{Y_0}]$ and the grid $\{(\mathsf{X0}+\mathsf{a}\cdot\mathsf{Dx}, \mathsf{y0}+\mathsf{b}\cdot\mathsf{Dy}) \mid \mathsf{a},\mathsf{b}\in\mathbb{R}\}$. The distances Dx and Dy of the grid points in the x- and y-directions depend considerably on the screen resolution: You might adapt them in the program source or — even better — change the size of the OPEN GEOMETRY window.

We recommend that you use a rather small and square window. Run an arbitrary OPEN GEOMETRY program, press <`Ctrl+W`> or choose the menu item

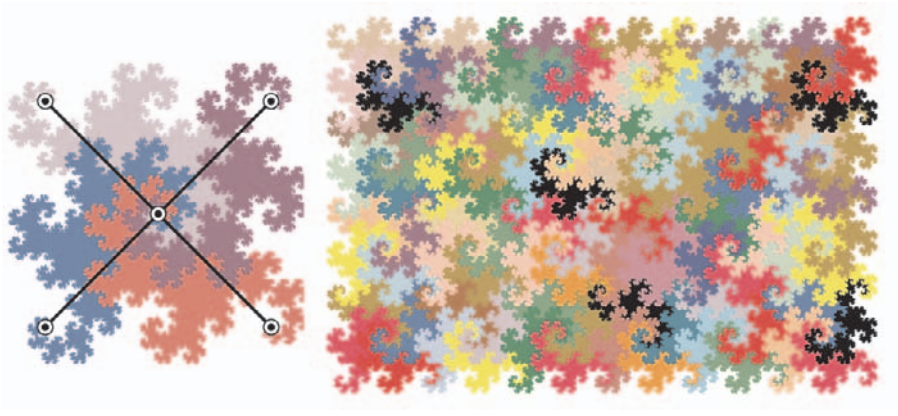

FIGURE 2.39. Paving the plane with Dragons.

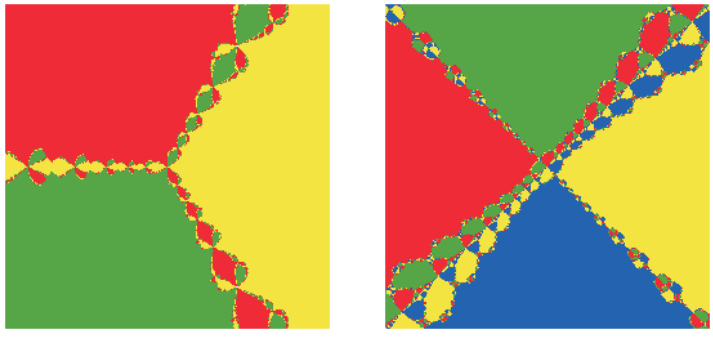

FIGURE 2.40. Two fractals generated by Newton's iteration.

"Image→Window→Window dimensions" to open a dialogue window and take, e.g., a width of 300 and an aspect ratio of 1/1. Then the default values in `"cubic_newton_fractal.cpp"` will produce a good picture in reasonable computing time.

In Init( ) we set the coefficients of the cubic polynomial $Ax^3 + Bx^2 + Cx + D$ and compute its root. We do not allow quadratic or linear polynomials, since we do not want to take care of different cases at the cost of computing time (Solution is a globally defined array of three elements of type *Complex*).

Listing from `"cubic_newton_fractal.cpp"`:

```
void Scene::Init( )
{
```

```
    A.Set( 1, 0 );
    B.Set( 0, 0 );
    C.Set( 0, 0 );
    D.Set( -1, 0 );

    if ( A.Magnitude( ) < 1e-7 )
       SafeExit( "Not a a cubic equation!" );

    CubicSolve( B/A, C/A, D/A, Solution );
}
```

In Draw( ) we write two loops to start with all values of z in question. We copy z to another complex number w and do the following while count < MaxCount:

Listing from "cubic_newton_fractal.cpp":

```
w = Newton( w );
if ( IsRoot( w, Solution[0], Eps ) )
{
    count += MaxCount;
    z.MarkPixel( Yellow );
}
else if ( IsRoot( w, Solution[1], Eps ) )
{
    count += MaxCount;
    z.MarkPixel( Red );
}
else if ( IsRoot( w, Solution[2], Eps ) )
{
    count += MaxCount;
    z.MarkPixel( Green );
}
count++;
```

Of course, Newton(w) is NEWTON's iteration $\mathsf{w} \mapsto \mathsf{w} - P(\mathsf{w})/P'(\mathsf{w})$ for the polynomial P. In addition, we use the IsRoot(...) to check whether the input parameter w is sufficiently close to a root of our polynomial. "Sufficiently close" means that w lies in a disk of radius Eps around the root. We didn't, however, use the Euclidean metric but the *Manhattan metric* to define this disk:

Listing from `"cubic_newton_fractal.cpp"`:

```
Boolean IsRoot( Complex z, Complex root, Real eps )
{
    return fabs( z.get_re( ) − root.get_re( ) ) +
           fabs( z.get_im( ) − root.get_im( ) ) < eps;
}
```

If w is close to a zero, we paint the corresponding pixel in the appropriate color. The global real Eps can sometimes be chosen rather big, which will considerably decrease the computation time. In our program we used Eps = 0.8.

Denote by $A(\zeta_i)$ the region of attraction for the zero ζ_i. Then Eps may be the supremum of all reals r such that the "Manhattan disk" with radius r and center ζ_i completely lies in $A(\zeta_i)$.

Figure 2.40 shows the output of the programs `"cubic_newton_fractal.cpp"` and `"quartic_newton_fractal.cpp"`. There we use the polynomials

$$x^3 - 1 \quad \text{and} \quad x^4 + ix^2 - 1,$$

respectively. ♢

Example 2.28. The Mandelbrot set
Another real classic is the *Mandelbrot set*. It can be defined in different ways, but for computer graphics, the following does the best job:

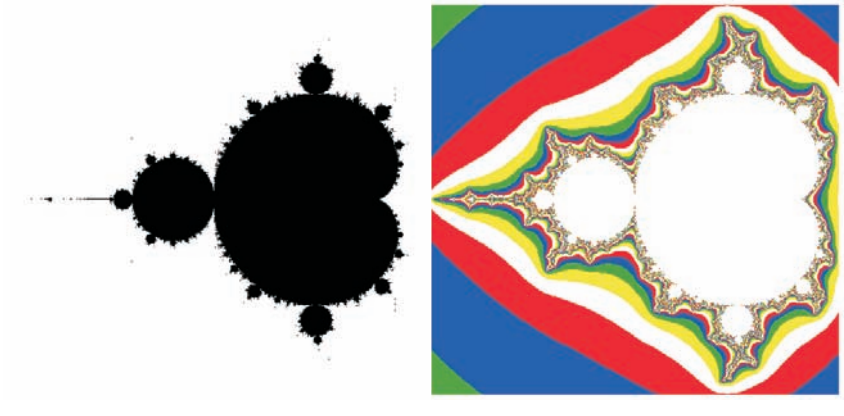

FIGURE 2.41. The Mandelbrot set M.

Let $c \in \mathbb{C}$ be a complex number. We define $f_c(z) := z^2 + c$, $f_c^0(z) := f_c(z)$, and for $k \in \mathbb{N}$, $f_c^{k+1}(z) := f \circ f_c^k(z)$ $(k \in \mathbb{N})$. That is, $f_c^k(z)$ is the k-th iterated function of $f_c(z)$. According to [7], the *Mandelbrot set* M can now be defined as

$$M := \left\{ c \in \mathbb{C} \mid \langle f_c^k(0) \rangle_{k \geq 1} \text{ is limited} \right\}.$$

This definition gives rise to an easy visualization of M on a computer screen (`"mandelbrot_set1.cpp"`; Figure 2.41). As in Example 2.27, we consider complex numbers from the intersection $\mathbb{D}$ of a rectangle $[\mathsf{X0}, \mathsf{X1}] \times [\mathsf{Y0}, \mathsf{Y1}]$ and a rectangular grid $\{\mathsf{X0} + \mathsf{a} \cdot \mathsf{Dx} + i(\mathsf{Y0} + \mathsf{b} \cdot \mathsf{Dy}) \mid \mathsf{a}, \mathsf{b} \in \mathbb{R}\}$. For each complex number $c \in \mathbb{D}$, we use the function IsInSet(...) to test whether it is in M or not:

Listing from `"mandelbrot_set1.cpp"`:

```
Boolean IsInSet( Complex c, int N, Real R )
{
    int i = 0;
    Complex c0 = c;
    while ( i < N )
    {
        c = c * c + c0;
        if ( c.Magnitude( ) > R )
            return false;
        i++;
    }
    return true;
}
```

We compute $f_c^k(0)$ and return false if its absolute value exceeds a certain limit R. If this does not happen for any $k \leq \mathsf{N}$, we assume that $c \in M$ and return true. The rest is done in Draw( ) (MaxCount = 100 and MaxRad = 100 are global constants):

Listing from `"mandelbrot_set1.cpp"`:

```
void Scene::Draw( )
{
    Real x = X0, y = Y0;
    Complex z;
    while ( x < X1 )
    {
        while ( y < Y1 )
        {
            z.Set( x, y );
            if ( IsInSet( z, MaxCount, MaxRad ) )
                z.MarkPixel( Black );
            y += Dy;
        }
        y = Y0;
        x += Dx;
    }
}
```

There exists a beautiful alternative visualization of M (`"mandelbrot_set2.cpp"`; Figure 2.41). Instead of painting the pixels of M, we can paint the pixels corresponding to $\mathbb{C} \setminus M$. The color depends on the first integer k_0 for which $f_c^{k_0}(0)$ exceeds the maximum radius MaxRad. All we have to do is to change the function IsInSet(...) a little:

Listing from `"mandelbrot_set2.cpp"`:

```
const int C = 5;
const Color Col[C] = { White, Yellow, Green, Blue, Red};

int IsInSet( Complex c, int N, Real R )
{
   int i = 0;
   Complex c0 = c;
   while ( i < N )
   {
      c = c * c + c0;
      if ( c.Magnitude( ) > R )
         return i % C;
      i++;
   }
   return 0;
}
```

It returns an integer value $i \in \{0, \ldots, C-1\}$, where C is the dimension of an arbitrary array of colors. The Draw( ) only needs needs a small change:

Listing from `"mandelbrot_set2.cpp"`:

```
void Scene::Draw( )
{
   Real x = X0, y = Y0;
   Complex z;
   int i;
   while ( x < X1 )
   {
      while ( y < Y1 )
      {
         z.Set( x, y );
         i = IsInSet( z, MaxCount, MaxRad );
         z.MarkPixel( Col[i] );
         y += Dy;
      }
      y = Y0;
      x += Dx;
   }
}
```

♢

2.5 Conics

Together with points and straight lines, conic sections belong to the most fundamental objects of two-dimensional geometry, whether Euclidian, affine, or projective. They appear in many contexts and despite investigation for 2500 years, interesting new properties are still being discovered.

The OPEN GEOMETRY classes *Conic* and *Conic3d* have been considerably improved and enlarged in OPEN GEOMETRY 2.0. In this chapter we will present their most important methods in action. For a complete listing the reader is referred to Chapter 6, page 446 and page 490.

There exist several ways of defining a conic section. Analytically speaking, it is the locus of all points in a plane that satisfy an equation of the type

$$c_1x^2 + c_2xy + c_3y^2 + c_4x + c_5y + c_6 = 0 \tag{6}$$

with respect to a Euclidean coordinate system. Each time you define a conic in OPEN GEOMETRY, the program computes the coefficients c_i of this equation. With their help the conic type (*ELLIPSE*, *PARABOLA*, *HYPERBOLA*, or *IRREGULAR*) is determined and the vertices, the center, the axis, the foci, the asymptotes, and the curve points are computed. Many other methods of Conic refer to equation (6) as well. It allows efficient, precise, and elegant computations.

The original way of implementing a conic in OPEN GEOMETRY is described in [14], Section 4.6.[15] In OPEN GEOMETRY 1.0 it was necessary to specify five conic points in order to define a conic:

```
P2d P[5];
P[0].Def( -5, 0 );
P[1].Def( 0, -3 );
P[2].Def( 5, 0 );
P[3].Def( 0, 3 );
P[4].Def( 4, 1.8 );
Conic conic;
int number_of_points = 150;
conic.Def( Black, number_of_points, P );
conic.Draw( THICK );
```

In the new version you have more possibilities. You can define a conic section by

1. one point plus two focal points,,
2. three points plus one focal point,
3. five tangents,

[15]The list of methods of the class *Conic* given there is no longer up to date! The new version has much more in store for you.

4. four points plus one tangent in one of them,
5. the center plus two end points of a pair of conjugate diameters,
6. the coefficients of the implicit equation (6).

For your convenience, we list the corresponding function headers from `"conic.h"`:

Listing from `"H/conic.h"`:

```
void Def( Color col, int numPoints, const P2d P[5] );
void Def( Color col, int numPoints, const P2d &P,
        const P2d &F1, const P2d &F2, TypeOfConic type );
void Def( Color col, int numPoints, const P2d &P,
        const P2d &Q, const P2d &R, const P2d &F );
void Def( Color col, int numPoints, const StrL2d t[5] );
void Def( Color col, int numPoints, const P2d P[3],
        const P2d &T, const StrL2d &t );
void Def( Color col, int numPoints, const P2d &M,
        const P2d &A, const P2d &B );
void Def( Color col, int numPoints, Real d[6] );
```

The use of these new defining methods is indicated in a first example. There we will define a conic section by three points and one focal point.

Example 2.29. Focal points and catacaustic
The sample program `"catacaustic.cpp"` shows an example of the definition of a conic section s by three points $P_0, P_1, P_2 \in s$ and one focal point E of s. There we want to illustrate the following interesting theorem (compare [4], [25]):

The catacaustic c of a pencil of lines $E(e)$ with respect to a reflecting curve r is the locus of all focal points $F \neq E$ of those conic sections that osculate r and have E as one focal point.

What needs to be done? We define a global vertex E of the pencil of lines. The reflecting curve r is a parameterized curve, in our case an ellipse. Furthermore, we use a global variable T that gives the current parameter of the curve point on r. It will be increased by a certain Δ in Animate( ). That is all we need to draw the osculating conic in Draw( ):

Listing from `"catacaustic.cpp"`:

```
void Scene::Draw( )
{
   ShowAxes2d( Black, −5, 10, −8, 8 );
   // get three neighboring points of the reflecting ... conic
   const Real eps = 0.001;
   P2d P[3];
   int i;
   for ( i = 0; i < 3; i++ )
      P[i] = ReflCurve.CurvePoint( T + ( i − 1 ) ∗ eps );
   // ...and define osculating conic section with one focal point E
   Conic osc_conic;
   osc_conic.Def( Red, 100, P[0], P[1], P[2], E );
   osc_conic.Draw( MEDIUM );

   // midpoint and second focal point of osc_conic
   P2d M = osc_conic.GetM( );
   P2d F = M + V2d( E, M );
   // the reflex of F on refl_curve
   P2d R = ReflCurve.CurvePoint( T );

   // draw the axes of osc_conic
   Real b = osc_conic.DistMC( );
   Real e = 0.5 ∗ E.Distance( F );
   Real m = maximum( osc_conic.DistMA( ), e );
   osc_conic.MajorAxis( ).Draw( Red, −1.5 ∗ m, 1.5 ∗ m, THIN );
   osc_conic.MinorAxis( ).Draw( Red, −1.5 ∗ b, 1.5 ∗ b, THIN );

   // draw the rest and mark the relevant points
   StraightLine2d( Gray, E, R, THIN );
   StraightLine2d( Gray, R, F, THIN );
   // GetCata(...) is a method of the class ParamCurve2d
   ReflCurve.GetCata( E, ReflCurve.u1, ReflCurve.u2, Catacaustic );
   Catacaustic.Draw( MEDIUM );
   ReflCurve.Draw( THICK );
   F.Mark( Red, 0.2, 0.1 );
   R.Mark( Blue, 0.2, 0.1 );
   E.Mark( Red, 0.2, 0.1 );
}
```

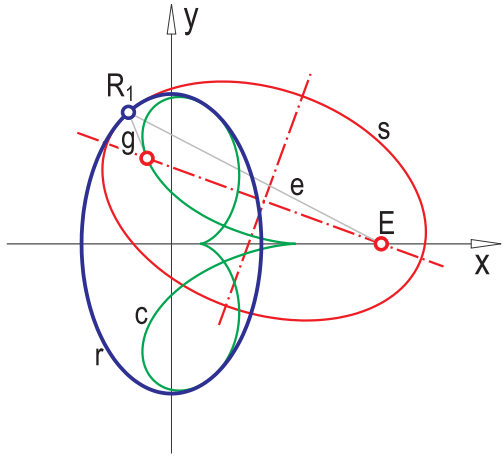

FIGURE 2.42. Output of the program `"catacaustic.cpp"`.

It is defined by three neighboring curve points and the focal point E. The rest of Draw( ) is routine. We make use of several *Conic* methods in order to get important conic points and draw important lines. Note that the drawing of the catacaustic is really very simple. We define and initialize a global variable Catacaustic of type *L2d* and use the GetCata(...) method of *ParamCurve2d*. ♢

In projective geometry, the *principle of duality* is well known. Its 2D version states that any theorem of projective geometry remains valid if we interchange the word "point" with "line" and the phrase "lies on" with "intersects." With the help of conic sections we can realize this principle. The central notion in this context is that of *polarity*.

Let c be a conic section in the projective plane.[16] The *outside* $\mathcal{O}(c)$ of c is defined as the set of all intersection points of two different and real tangents of c. That is, any point outside c is incident with two real conic tangents.

We consider a point $P \in \mathcal{O}(c)$. The conic tangents through P are p_1 and p_2, respectively (Figure 2.43). Their points of tangency span a straight line p that is called the *polar of P with respect to c*. Reversed, P is called the *pole of the straight line p*.

So far, we can associate a straight line p with each point $P \in \mathcal{O}(c)$ and a pole with each straight line that intersects c in two real points. The line p intersects

[16]If you are not familiar with the basic concepts of projective geometry, you can refer to Section 5.2 and read our short introduction to that topic, or you can read the following paragraphs without being too critical as far as range and image of certain maps are concerned.

the conic c in two real points. Now we extend this relation between points and lines to the other points. Let Q be a point in the *interior* $\mathcal{I}(c) := \mathbb{P}^2 \setminus (\mathcal{O}(c) \cup c)$ of c (Figure 2.43; there, we assumed, without loss of generality, that Q lies on p).

The point Q is incident with two straight lines p and t that intersect c in real points. Their respective poles P and T span a straight line q that will be called the *polar of Q with respect to c.*[17] Finally, if a point R is located on the conic, we define its polar as the conic tangent r in R.

Now, the conic c induces a bijection between the point set Π and the line set Λ of the projective plane $\mathbb{P}^2$. This bijection is called the *polarity of c*. It is a realization of the principle of duality because of the following theorem:

If three points P, T, and U are situated on a straight line q, their respective polars are concurrent in the pole Q of q.

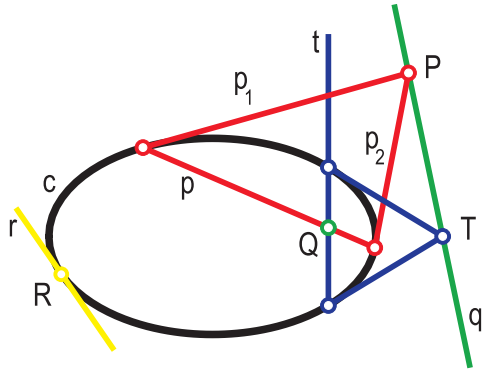

FIGURE 2.43. Poles and polars of a conic section c.

We visualize the above considerations in an OPEN GEOMETRY program:

Example 2.30. Pole and polar
In `"pole_and_polar.cpp"` we display a conic section C together with a couple of straight lines and points that correspond in the polarity on C:

Listing from `"pole_and_polar.cpp"`:

```
#include "opengeom.h"

Conic C;
V2d Dir; // direction vector for animation
```

[17] q is well-defined; i.e., it does not depend on the special choice of p and t.

```
void Scene::Init( )
{
   // define conic through a pair of conjugate diameters.
   C.Def( Black, 100, Origin2d, P2d( 5, 0 ), P2d( 0, 3 ) );
   Dir.Def( 0, 1 );
}
void Scene::Draw( )
{
   // define point P and determine its polar p
   P2d P;
   P.Def( 6, 5 );
   StrL2d p;
   p = C.GetPolar( P );

   // determine point on polar and inside conic
   // and get its polar
   P2d S1, S2;
   if ( !C.SectionWithStraightLine( p, S1, S2 ) )
      SafeExit( "P is outside the conic!" );
   const Real f = 0.8;
   P2d Q;
   Q = f * S1 + ( 1 - f ) * S2;
   StrL2d q;
   q = C.GetPolar( Q );
   q.Draw( Green, -10, 10, MEDIUM );

   // get some line through Q and find its pole T;
   // T must be situated on the polar p of P
   StrL2d t;
   t.Def( Q, Dir );
   P2d T;
   T = C.GetPole( t );
   ...
}
void Scene::Animate( )
{
   Dir.Rotate( 1.2 );
}
```

We use one of Open Geometry's new methods for the definition of the conic in Init( ): It is given by its center plus end points of a pair conjugate diameters (in our case, the end points happen to be the conic vertices).

In Draw( ) we define an arbitrary point $\mathsf{P} \in \mathcal{O}(\mathsf{C})$ and determine its polar p by using the GetPolar(...) method of *Conic*. We test whether p and C have two real points in common (they have only if P is located in the outside of C) and store them in S1 and S2, respectively.

Next, we define the point $\mathsf{Q} \in \mathsf{p} \cap \mathcal{I}(\mathsf{C})$ and get its polar q by means of GetPolar(...). It must be incident with P. Furthermore, we determine the pole T of a straight line t through Q. The direction of t will be changed in Animate( ), which causes T to travel along the polar q of Q. ♢

With the help of polarity and/or the principle of duality, one can derive certain geometric theorems almost immediately. For example, Brianchon's Theorem (1806) is an immediate consequence of Pascal's Theorem (1640):

Example 2.31. Pascal and Brianchon
The theorems of Pascal and Brianchon are both visualized in Figure 2.44. They can be formulated as follows:

Theorem (Pascal): *Let A_i and B_i be two sets of three points each on a conic section c $(i = 0, 1, 2)$. Then the three intersection points $S_i := [A_j, B_k] \cap [A_k, B_j]$ $(i, j, k \in \{0, 1, 2\}$ pairwise different$)$ are collinear.*

Theorem (Brianchon): *Let a_i and b_i be two sets of three tangents each of a conic section c $(i = 0, 1, 2)$. Then the three connecting lines $s_i := [a_j \cap b_k, a_k \cap b_j]$ $(i, j, k \in \{0, 1, 2\}$ pairwise different$)$ have a common point.*

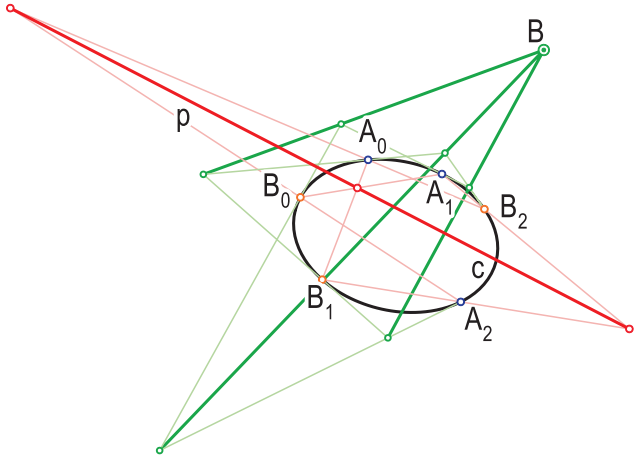

FIGURE 2.44. The theorems of Pascal (1640) and Brianchon (1806).

We produced Figure 2.44 with the help of "pascal_brianchon.cpp". There we globally define a conic C, six of its points, and six of its tangents:

Listing from "pascal_brianchon.cpp":

```
Conic C;
P2d A[3], B[3];      // six points on conic
StrL2d a[3], b[3];   // six tangents of conic

// pulsing reals for animation
PulsingReal Rx[5], Ry[5];
```

We want to demonstrate both theorems with the help of a little animation. Therefore, the conic as well as the conic points and tangents will change with every new frame. In order to achieve this, we use additional instances of the Open Geometry class *PulsingReal*. Only they can be initialized in Init( ):

Listing from "pascal_brianchon.cpp":

```
void Scene::Init( )
{
   // initialize pulsing reals
   Rx[0].Def( -4.5, 0.02, -3.5, -5.5, HARMONIC );
   Ry[0].Def( 2, 0.01, 1.5, 4, HARMONIC );
   ...
   Rx[4].Def( 0, 0.02, -2, 1, HARMONIC );
   Ry[4].Def( -5, 0.03, -3.5, -6, HARMONIC );
}
```

The remaining elements have to be defined in Draw( ) (otherwise, there would not be any animation). The essential part is this:

Listing from `"pascal_brianchon.cpp"`:

```
// define conic
P2d P[5];
int i;
for ( i = 0; i < 5; i++ )
    P[i].Def( Rx[i].Next( ), Ry[i].Next( ) );
C.Def( Black, 100, P );

// initialize points on conic
A[0] = P[0], A[1] = P[1], A[2] = P[2];
B[0] = P[3], B[1] = P[4];
B[2] = A[2];
B[2].Reflect( C.MajorAxis( ) );

// initialize tangents of conic
for ( i = 0; i < 3; i++ )
{
    a[i] = C.GetPolar( A[i] );
    b[i] = C.GetPolar( B[i] );
}

// three points on PASCAL axis
P2d S[3];
S[0] = StrL2d( A[1], B[2] ) * StrL2d( A[2], B[1] );
S[1] = StrL2d( A[0], B[2] ) * StrL2d( A[2], B[0] );
S[2] = StrL2d( A[1], B[0] ) * StrL2d( A[0], B[1] );

// point of BRIANCHON
P2d T[3], U[3];
T[0] = a[1] * b[2], U[0] = a[2] * b[1];
T[1] = a[0] * b[2], U[1] = a[2] * b[0];
T[2] = a[1] * b[0], U[2] = a[0] * b[1];
P2d Q = StrL2d( T[0], U[0] ) * StrL2d( T[1], U[1] );

C.Draw( THICK );
```

We define the conic with the help of five "pulsing" points P[i]. Their definition ensures that they will slowly wander over the screen during the animation. We copy them to the points A[i] and B[i], respectively. Only $B[2]$ needs a special treatment in order to guarantee that it is located on C.

The class *Conic* does not know a Tangent(...) method. It is not necessary, because GetPolar(...) does the job. We use this to initialize the conic tangents. Now we construct the three points S[i] on the PASCAL axis and BRIANCHON's point Q as described in the above theorems.

In the remaining part of Draw( ) we plot a large number of straight lines on the screen and mark important points. It is of little interest, and we do not display it here. Note, however, a little detail: In order to draw the PASCAL axis, we write the following lines:

Listing from "pascal_brianchon.cpp":

```
StraightLine2d( Red, S[0], S[1], THICK );
StraightLine2d( Red, S[1], S[2], THICK );
```

This ensures that, independent of the order of the points S[i] (which may change during the animation!), exactly the line segment between all three points S[i] will be displayed. ♢

Sometimes, it is better to use an instance of *ParamCurve2d* instead of the class *Conic*. The following example illustrates this.

Example 2.32. Conic caustics
We have already talked about caustics of reflection (compare Example 2.6). Reflecting the rays of a pencil of lines with vertex E on a plane curve c yields a hull curve c_1, the *catacaustic* (or simply *caustic*) of c with respect to the pole E. Of course, a ray of light may be reflected not only once but twice or more often. The hull curve after n reflections is called *catacaustic* or *caustic of order n*.

It is very easy to visualize the caustics of higher order if c is a conic section. The reason for this is that we can easily compute the reflection of a ray of light on the conic. In "conic_caustic.cpp" we introduce global constant reals A and B and a global constant Type that stores the type of the reflecting conic c (*ELLIPSE*, *PARABOLA*, or *HYPERBOLA*). Then we parameterize c:

Listing from "conic_caustic.cpp":

```
P2d ConicPoint( Real t )
{
  if ( Type == ELLIPSE )
```

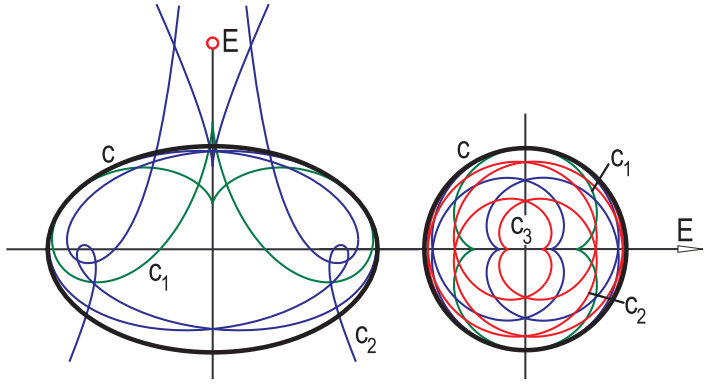

FIGURE 2.45. The caustics of first and second order of an ellipse (left) and the caustics up to order three of a circle (right). In the second case, the pole E is a point at infinity.

```
        return P2d( A * cos( t ), B * sin( t ) );
    if ( Type == HYPERBOLA )
        return P2d( A / cos( t ), B * tan( t ) );
    else // parabola
    {
        const Real factor = -B - B * sin( t );
        return P2d( A * B * cos( t ) / factor,
                    2 * B * B * sin( t ) / factor );
    }
}
```

Note that the suggested parametric representations for parabola and hyperbola are not standard! Compared to the more frequent parameterized equations

$$\begin{pmatrix} t \\ t^2 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} a\cosh(t) \\ b\sinh(t) \end{pmatrix},$$

respectively, they have the several advantages:

1. They require the same parameter interval $[0, 2\pi]$ for all three conic types, which is convenient for a program that treats all types equally.
2. Our parameterizations are *periodic*. This is an advantage if we vary a point on the conic during the animation.
3. We can parameterize both branches of a hyperbola with a single equation. Additionally, the distribution of points on parabola and hyperbola is quite good.

Of course, the use of two design parameters A and B is redundant in the parabolic case. We use them for reasons of uniformity. In Init( ) we compute five conic points and define c = ReflConic:

Listing from "conic_caustic.cpp":

```
void Scene::Init( )
{
  P2d P[5];
  int i;
  for ( i = 0; i < 5; i++ )
    P[i] = ConicPoint( i );
  ReflConic.Def( Black, 100, P );
}
```

At first sight, the use of the function ConicPoint(...) seems to be a bit exaggerated, but we will be able to employ it very well in the central part of "conic_caustic.cpp". This part consists of a function to reflect a straight line on c. The input parameters are the center E of the light rays, the number n of reflections, and the curve parameter t.

Listing from "conic_caustic.cpp":

```
StrL2d ReflectLine( P2d E, int n, Real t )
{
  P2d C = ConicPoint( t );
  StrL2d s( C, E );
  StrL2d c = ReflConic.GetPolar( C );
  s.Reflect( c );
  int i;
  P2d S1, S2;
  for ( i = 1; i < n; i++ )
  {
    ReflConic.SectionWithStraightLine( s, S1, S2 );
    if ( S2 == C )
    {
      S2 = S1;
      S1 = C;
    }
    s.Def( S1, S2 );
    c = ReflConic.GetPolar( S2 );
    s.Reflect( c );
    C = S2;
  }
  return s;
}
```

We connect E and a conic point C = C(t) by a straight line s and reflect it on c. In order to get the tangent of c in C, we use OPEN GEOMETRY's GetPolar(...) method of the class *Conic*. Next, we determine the intersection points S1 and S2 of the reflected ray s and the conic c. Actually, s is an oriented straight line, and we use it to arrange S1 and S2 in a certain order. We reflect s on the tangent of c in S2 and repeat the whole procedure until we reach n reflections. The returned straight line is a tangent of the caustic c_n of order n, and we can implement c_n as a class curve:

Listing from `"conic_caustic.cpp"`:

```
class MyCaustic: public ClassCurve2d
{
public:
   StrL2d Tangent( Real t )
   {
      return ReflectLine( E, refl_numb, t );
   }
   int refl_numb;
   void Def( Color c, int m, Real umin, Real umax, int n )
   {
      refl_numb = n;
      ClassCurve2d::Def( c, m, umin, umax );
   }
};
MyCaustic Caustic[N];
```

We use a global constant DrawAll of type *Boolean* to display all caustics c_i up to a certain order together in one image or to draw them separately in each new frame. In Draw( ) we write the following:

Listing from `"conic_caustic.cpp"`:

```
int k = FrameNum( ) % N;
Caustic[k].Def( Col[k%9], k * 200, -PI, PI, k );

if ( DrawAll )
{
   int i;
   for ( i = 0; i < k; i++ )
      Caustic[i].Draw( THIN, 20 );
}
else
   Caustic[k].Draw( THIN, 20 );
```

Finally, we mark the eye point E and draw reflecting conic and coordinate axes. The output of the program can be seen in Figure 2.45. ♢

Conics in 3-space are described by the OPEN GEOMETRY class *Conic3d*. Actually, 3D conics are not a good topic for a chapter on 2D graphics, but since *Conic3d* is derived from *Conic*, we will make an exception. Internally, an instance of *Conic3d* is determined as follows:

1. From the input data (certain points or lines) we calculate the conic's supporting plane σ.[18]
2. We project the input data orthogonally into one of the coordinate planes (usually *XYplane*). If the intersection angle of σ and *XYplane* is too close to 90°, we take *YZplane* or *XZplane* instead.
3. From the projected input data, we compute a 2D conic and project its points back into σ.

Most of the usual conic methods like GetA( ), DistMA( ) and GetPole(...) refer to the 3D conic. The only exception is WriteImplicitEquation( ): It describes the projection cylinder.

The final example in this chapter deals with conics in 3-space:

Example 2.33. Focal conics
Two conics c_1 and c_2 in 3-space are called *a pair of focal conics* if:

1. their supporting planes are orthogonal;
2. the vertices of c_1 are the focal points of c_2 and vice versa.

A pair of focal conics consists of either one ellipse and one hyperbola or two parabolas. We will display both types in the OPEN GEOMETRY program `"focal_conics.cpp"`. To begin with, we implement the pair consisting of ellipse and hyperbola. The Init( ) part reads as follows:

Listing from `"focal_conics.cpp"`:

```
P3d P;
P.Def( 0, -4, 0 );
E1.Def( 4, -9, 0 );
E2.Def( -4, -9, 0 );
```

[18] Note that we do not check that all points and lines are really coplanar! The conic's supporting plane σ is calculated from some input data only. The remaining points and lines will be projected into σ.

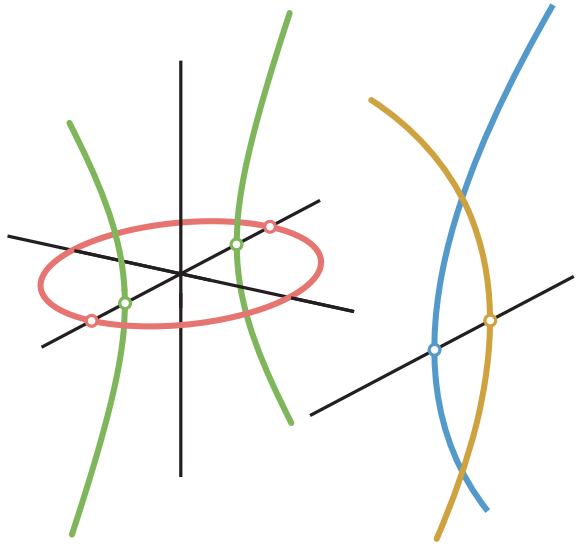

FIGURE 2.46. Two pairs of focal conics (output of "focal_conics.cpp").

```
FocalConic1.Def( Red, 100, P, E1, E2, ELLIPSE );

F1 = FocalConic1.GetA( );
F2 = FocalConic1.GetB( );

V3d v = FocalConic1.GetC( ) - FocalConic1.GetM( );
P = F1 + 2 * FocalConic1.DistMC( ) / E1.Distance( E2 ) * v;
P.Rotate( StrL3d( E1, E2 ), 90 );

TypeOfConic type2 = ( FocalConic1.type( ) == ELLIPSE ) ?
    HYPERBOLA : ELLIPSE;
FocalConic2.Def( Green, 100, P, F1, F2, type2 );
```

The code is quite straightforward and easy to understand. The focal conics FocalConic1 and FocalConic2 as well as their focal points E1, E2, F1, and F2 are globally declared. We define the first focal conic FocalConic1 by its focal points and a third point P. In order to avoid ambiguities, we must specify the conic type as well.

Here, F1 and F2 are the vertices of the first conic and the focal points of FocalConic2. All we need is a third point of the second conic. We get it with the help of a simple formula that requires the first conic's semiaxis length and eccentricity. The type of FocalConic2 is chosen with respect to the type of FocalConic1.

Now we continue by implementing the two focal parabolas FocalParabola1 and FocalParabola2 in Init( ). Their focal points are P1 and P2, respectively.

Listing from "focal_conics.cpp":

```
P1.Def( -2, 4, 0 );
P2.Def( 2, 4, 0 );

P3d Q[5];
Q[0] = P2;
Real dist = P1.Distance( P2 );
V3d w;
w.Def( 0, 0, 1 );
Q[1] = P1 + 2 * dist * w;
Q[2] = P1 - 2 * dist * w;
v = 0.5 * ( P2 - P1 );
dist *= sqrt( 2 );
Q[3] = P1 + v + dist * w;
Q[4] = P1 + v - dist * w;

FocalParabola1.Def( Blue, 100, Q );

Plane p;
p.Def( Rod3d( NoColor, P1, P2 ) );
int i;
for ( i = 0; i < 5; i++ )
    Q[i].Reflect( p );

FocalParabola2.Def( Yellow, 100, Q );
```

This time it is convenient to define the conics via five points. We choose them symmetric with respect to the straight line P1P2. The points of FocalParabola2 are obtained through a reflection on the parabola's plane of symmetry p.

In Draw( ) we display the four conics together with their axes and focal points (vertices, respectively). The output of the program is displayed in Figure 2.46.

Listing from "focal_conics.cpp":

```
void Scene::Draw( )
{
  FocalConic1.Draw( THICK );
  FocalConic2.Draw( THICK );
  FocalConic1.MajorAxis( ).Draw( Black, −10, 10, THIN );
  FocalConic1.MinorAxis( ).Draw( Black, −8, 8, THIN );
  FocalConic2.MinorAxis( ).Draw( Black, −15, 15, THIN );

  FocalParabola1.Draw( THICK );
  FocalParabola2.Draw( THICK );
  FocalParabola1.MajorAxis( ).Draw( Black, −10, 10, THIN );

  Zbuffer( false );
  E1.Mark( Green, 0.2, 0.1 );
  E2.Mark( Green, 0.2, 0.1 );
  F1.Mark( Red, 0.2, 0.1 );
  F2.Mark( Red, 0.2, 0.1 );
  P1.Mark( Yellow, 0.2, 0.1 );
  P2.Mark( Blue  , 0.2, 0.1 );
}
```

Focal conics have remarkable properties. For example, if you choose a point on the first conic and connect it with all points of the second conic, you will always get a *cone of revolution* (wouldn't this be a nice idea for an Open Geometry program?). An interesting relation of *focal quadrics* is the content of Example 3.14. ♢

2.6 Splines

Bézier curves

In computer-aided geometric design (CAGD), Bézier curves and Bézier surfaces play an important role. Open Geometry 2.0 provides the six new classes *BezierCurve2d*, *BezierCurve3d*, *RatBezierCurve2d*, *BezierCurve3d*, *BezierSurface*, and *RatBezierSurface*. In this chapter we are going to talk about *BezierCurve2d* and *RatBezierCurve2d*.

Corresponding 2D and 3D classes are *identical in the* Open Geometry *sense*. That is, we implemented the 3D classes by replacing all occurrences of "2d" by "3d". Therefore, the classes *BezierCurve3d* and *RatBezierCurve3d* are not explicitly explained in this book. In the following, we will deal with 2D Bézier

curves only. If you can handle them, you will be able to handle Bézier curves in three dimensions as well.

A Bézier curve c is usually defined by its *control points* $B_0, \ldots, B_n$. The corresponding polygon $P = \langle B_0, \ldots, B_n \rangle$ is called the *control polygon*; the integer n is the *degree of* c. In Figure 2.47 you can see why Bézier curves are so popular: *The shape of the control polygon and the Bézier curve are very similar*. To be more accurate, Bézier curves have the following properties:

- The start point S of c is the first vertex of P, the end point E is the last vertex.
- The first and last edge of P are tangent to c in S and E, respectively.
- The Bézier curve b lies in the convex hull of P (*convex hull property*; in Figure 2.47, this area is shaded).
- An arbitrary straight line s intersects P in at least as many points as b (*variation diminishing property*).

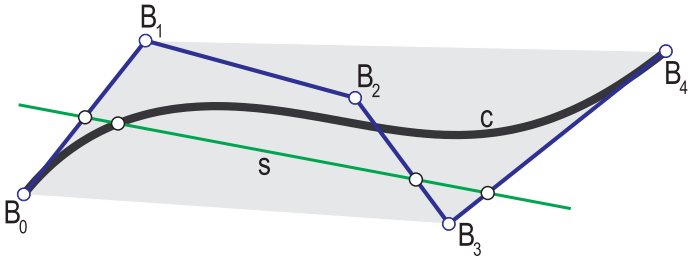

FIGURE 2.47. A Bézier curve of order four.

Thus, it is easy to control the shape of a Bézier curve interactively. The user can simply change certain control points and adapt the curve to his/her needs. In contrast to spline curves, one must, however, take into account that the position of a single control point affects all points of c.

Bézier curves have many additional properties that are of importance in CAGD. We present the most fundamental of them along with the corresponding OPEN GEOMETRY implementations. The file `"bezier_curve.cpp"` is a good templet for reference.

There exists a well-known algorithm to determine a curve point C of c: DE-CASTELJAU's algorithm. It works as follows:

1. Choose a real $u_0 \in [0, 1]$.[19]

[19]It is common practice (but not absolutely necessary) to restrict the parameter interval to $[0, 1]$.

2. Divide each edge of the control polygon into segments of affine ratio $u_0 : (1 - u_0)$. This yields a polygon P_1 of $n - 1$ edges.

3. Proceed in the same way with the polygon P_1. This yields a sequence $\langle P_1, P_2, \ldots, P_n \rangle$ of polygons. P_n consists of one single point: a curve point of c.

Figure 2.48 illustrates this process.

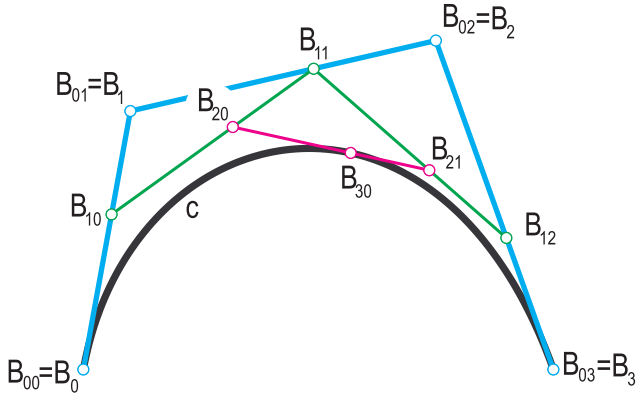

FIGURE 2.48. The algorithm of DeCasteljau.

In Open Geometry, a Bézier curve is a class derived from *ParamCurve2d*. The CurvePoint(...) function calls the private method DeCasteljau(...).[20] Thus, in order to get a Bézier curve in Open Geometry, we have to declare a global variable of type *BezierCurve2d* and define it in Init( ):

Listing from `"bezier_curve.cpp"`:

```
BezierCurve2d BezierCurve;
void Scene::Init( )
{
    P2d P[4];
    P[0]( -10, -7 );
    P[1]( -8, 4 );
    P[2]( 5, 7 );
    P[3]( 10, 7 );
    BezierCurve.Def( Black, 200, 4, P );
}
```

[20]The mathematical parametric representation of Bézier curves (it uses *Bernstein polynomials*) is of no interest to us in this book: we simply do not need it.

The Bézier curve is of order three, i.e., there exist four control points. Instead of using an array P of points, you could alternatively use an object of type *Coord2dArray* for the definition. In Draw( ) we draw the curve and control polygon and mark the control points:

Listing from `"bezier_curve.cpp"`:

```
void Scene::Draw( )
{
  BezierCurve.Draw( VERY_THICK );
  BezierCurve.DrawControlPolygon( Blue, THICK );
  BezierCurve.MarkControlPoints( Blue, 0.15, 0.1 );
}
```

If we restrict the parameter value to $[0, 1]$, DeCasteljau's algorithm is fast and numerically stable. It is invariant to affine transformation, and it yields a field of very useful points B_{ij}. In our example, the point B_{30} is a curve point of c, while B_{20} and B_{21} determine the tangent of c in B_{30} (compare Figure 2.48).

The polygon $\langle B_{00}, B_{10}, B_{20}, B_{30}\rangle$ from DeCasteljau's algorithm is the control polygon of a Bézier curve c_l that is identical to the part of c between B_{00} and B_{30}. The polygon $\langle B_{30}, B_{21}, B_{12}, B_{03}\rangle$ determines the part c_r of c between B_{30} and B_{03} (Figure 2.49). This possibility of *splitting* a Bézier curve is very important in practical applications. It is more or less a linear parametric transformation in terms of control points.

If we apply the same algorithm with a parameter value from $\mathbb{R} \setminus [0, 1]$, we *enlarge* the Bézier curve in one direction or the other. This process is illustrated on the right-hand side of Figure 2.49.

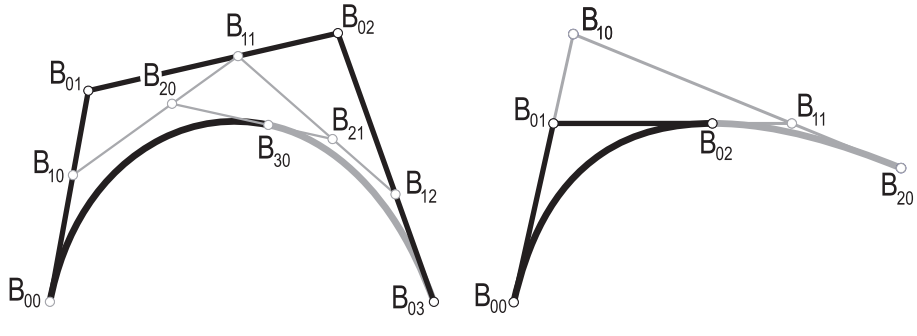

FIGURE 2.49. The splitting (left) and enlarging (right) of a Bézier curve.

The corresponding Open Geometry routine is *BezierCurve2d*::Split(*Real* u, *Boolean* second_half). For example, the line

```
BezierCurve[3].Split( 0.5, true );
```

redefines the control polygon and the curve itself. As a result, you get the part belonging to $u \in [0.5, 1]$ of the original curve. If the second parameter is false (default value), the part corresponding to $[0, 0.5]$ will be drawn. Note that the order of the control points will be reversed by Split(...) in the first case (i.e., the original end point of the Bézier curve will be the new start point). You can undo this by calling ReverseOrder( ) immediately after splitting the curve.

A Bézier curve of degree n can always be defined as a Bézier curve of degree $n+1$ or — more generally speaking — of degree $n+\nu$ ($\nu \in \mathbb{N}$). The construction of the control points of the degree elevated curve is displayed in Figure 2.50. It involves the repeated subdivision of the edges into segments of affine ratio $k : n-k+1$, $(k = 0, \ldots, n+1)$.

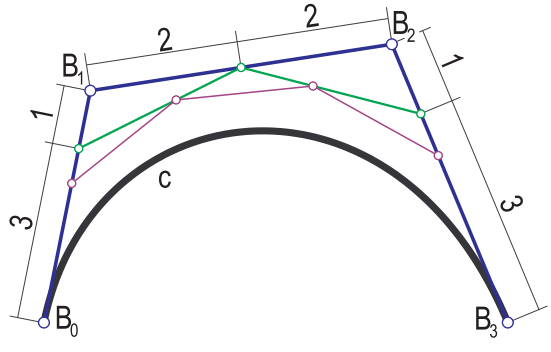

FIGURE 2.50. The degree of the Bézier curve is elevated twice.

Open Geometry provides the method ElevateDegree(...) to do this. The line

```
BezierCurve.ElevateDegree( 3 );
```

will, e.g., elevate the degree of the Bézier curve by 3.

Before presenting a first example we have to mention that *rational Bézier curves*, too, are available in Open Geometry. In contrast to ordinary (integral) Bézier curves, rational Bézier curves offer the possibility of assigning *weights* to the control points. Figure 2.51 shows an example of this. We display two rational Bézier curves with identical control polygon but different weights.

Many properties and algorithms of integral Bézier curves remain valid for rational Bézier curves as well. The reason for this lies in the geometric interpretation of *rational Bézier curves as central projections of integral Bézier curves*:

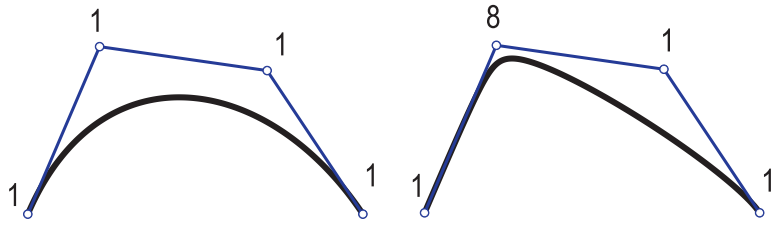

FIGURE 2.51. An integral and a rational Bézier curve.

Let c be a rational Bézier curve with control points $P_i(x_i, y_i)$ and weights w_i. We can embed them in $\mathbb{R}^2$ by assigning the z-coordinate 1 to them; i.e., we map (x, y) to $(x, y, 1)$. Furthermore, we introduce the points Q_i according to

$$Q_i = \begin{cases} (w_i x_i, w_i y_i, w_i) & \text{if } w_i \neq 0, \\ (x_i, y_i, 0) & \text{if } w_i = 0. \end{cases}$$

The points P_i and Q_i are located on a straight line through the origin. The control points Q_i define an integral Bézier curve d in 3-space, and *c is the central projection of d from O onto the plane $z = 1$* (Figure 2.52).

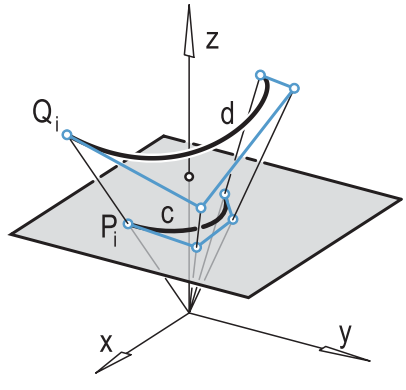

FIGURE 2.52. Rational Bézier curves are obtained as central projections of integral Bézier curves.

For rational Bézier curves in 3D, an analogous procedure is possible. The auxiliary curve d lies, however, in 4-space. This geometric interpretation allows an immediate transformation of DeCasteljau's algorithm to rational Bézier curves. The algorithms for degree elevation and curve splitting work as well.

Note that the family of rational Bézier curves is much vaster than that of integral Bézier curves. For example, an integral Bézier curve of order two is always a parabola, while a rational Bézier curve of order two is the central projection of a parabola, i.e., a conic section.

In this book, rational Bézier curves will not be described in detail. For information on defining methods the reader is referred to page 470 in Chapter 6. Furthermore, `"rat_bezier_curve.cpp"` is a good reference if you want to learn the usage of the class *RatBezierCurve2d*. Note that only simple modifications are necessary to make the subsequent examples work for rational Bézier curves as well.

In diverse applications of Bézier curves it is necessary to compose a curve of pieces of Bézier arcs. The different parts have to be connected in a "smooth" way. If two arcs have a common tangent at the end point where they meet, they are said to be of GC^1*-continuity.*[21] If they share the osculating circle as well, they are of GC^2-continuity, etc. In the following example we present a few (rather simple) algorithms for the construction of transition curves of GC^n continuity.

Example 2.34. Bézier curves of GC^n-continuity
We want to show how to construct a transition curve of GC^n-continuity to two given Bézier curves c and d. By $\vec{c}(u)$ and $\vec{d}(u)$ we will denote the DeCasteljau parametric representations of c and d, respectively. The control points of c and d will be $C_0, \ldots, C_m$ and $D_0, \ldots, D_n$, respectively. We want to construct a Bézier curve $s \ldots \vec{s}(u)$ that fulfills

$$\vec{c}_i(1) = \vec{s}_i(0) \quad \text{and} \quad \vec{s}_i(1) = \vec{d}_i(0)$$

for some integer k and $i = 0, \ldots, k$.[22] The higher the integer k is, the "smoother" the transition will be. The value $k = 0$ means that the end point of c and the starting point of s are identical while $k = 1$ yields identical tangents in this point, $k = 2$ identical osculating circles, etc. Analogous statements are true of the end point of s and the starting point of d. In order to solve the problem, we need the following general result on Bézier curves:

The i-th derivative of a Bézier curve in the start point (end point) depends on the first (last) i control points only.

Thus, the problem for $k = 1$ has a unique solution of degree two. The control points S_0, S_1, and S_2 of the transition curve s are determined by

$$S_0 = C_m, \quad S_1 = [C_m, C_{m-1}] \cap [D_0, D_1], \quad S_2 = D_0$$

(compare Figure 2.53, the output of `"gc_1-continuity.cpp"`).

In order to ensure GC^2-continuity, we need a transition curve of degree 4. The solution is, however, no longer unique. The osculating circle of c in the start point C_0 does not change if we translate the control point C_2 in the direction of the straight line $[C_0, C_1]$ (Figure 2.54).

[21] The letters "G" and "C" stand for *Geometric Continuity.*
[22] In this formula the subscript i denotes the i-th derivative of the vector function.

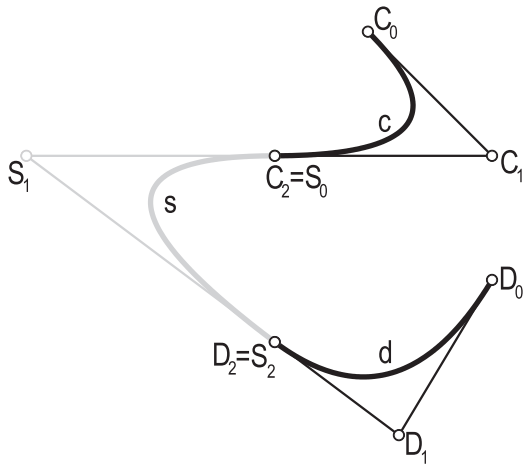

FIGURE 2.53. C^1-continuous transition curve between two Bézier curves c and d.

There are many possible configurations of the control points C_0, C_1, C_2 that yield the same curve c. Varying the point C_2 as described above gives a whole family of osculating Bézier curves. Still, we can use this to create GC^2-continuations in a very simple way (`"gc_2-continuity.cpp"`).

In Init( ) we initialize two arbitrary Bézier curves FirstCurve and SecondCurve. Then we declare two identical *local* Bézier curves:

Listing from `"gc_1-continuity.cpp"`:

```
BezierCurve2d first_curve, second_curve;
int n1 = FirstCurve.GetPNum( );
int n2 = SecondCurve.GetPNum( );
first_curve.Def( Green, 200, n1, P );
second_curve.Def( Blue, 200, n2, Q );
```

We make sure that their degree is at least 3 and "split" them at a parameter value with absolute value greater than 1.[23]

[23]Here we have two degrees of freedom. This indicates that the problem doesn't have a unique solution.

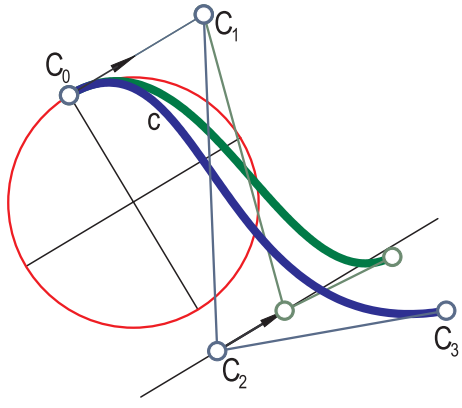

FIGURE 2.54. Two Bézier curves with a common osculating circle.

Listing from "gc_1-continuity.cpp":

```
if ( n1 <= 2 )
    first_curve.ElevateDegree( 3 - n1 );
if ( n2 <= 2 )
    second_curve.ElevateDegree( 3 - n2 );

first_curve.Split( 2.0 );
second_curve.Split( -2.0 );
```

The effect is that the curves are actually enlarged in one direction or the other. Next we split the curves again at the reciprocal parameter value and take the appropriate half:

Listing from "gc_1-continuity.cpp":

```
first_curve.Split( 0.5, true );
second_curve.Split( 0.5 );
first_curve.ReversePolygon( );
```

The local Bézier curves are now GC^∞–continuations of c and d, respectively (the second curve only up to a reversal of the order of the control points; Figure 2.55). This means that we can take the first and last three control points to build a Bézier curve of GC^2–continuity. If we vary the third point as described, we need only a transition curve of order four:

Listing from "gc_1-continuity.cpp":

```
P2d R[5];
R[0] = first_curve.GetControlPoint( 0 );
R[1] = first_curve.GetControlPoint( 1 );
R[3] = second_curve.GetControlPoint( 1 );
R[4] = second_curve.GetControlPoint( 0 );

P2d A, B;
A = first_curve.GetControlPoint( 2 );
B = second_curve.GetControlPoint( 2 );
StrL2d s1, s2;
s1.Def( A, V2d( R[0], R[1] ) );
s2.Def( B, V2d( R[3], R[4] ) );
R[2] = s1 * s2;
TransitionCurve.Def( Red, 200, 5, R );
```

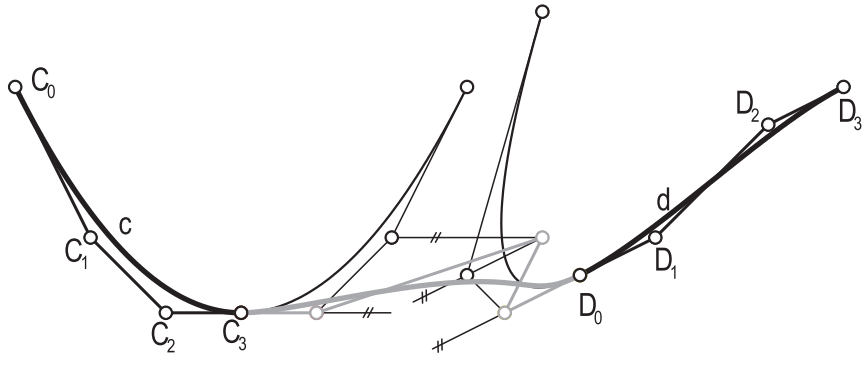

FIGURE 2.55. A GC^2–transition curve t to c and d.

Of course, we can create GC^n transition curves for an arbitrary $n \in \mathbb{N}$ as well. We determine GC^∞ continuations of c and d as above, elevate their degree if necessary, and take n points from each continuation curve to build the control polygon of the transition ("c_n-continuity.cpp"). Note, however, that this algorithm does not produce the solution of lowest degree to the problem. ♢

A great advantage of Bézier curves is the possible intuitive design of curves. You start with a control polygon that resembles the shape you want to create. Then you vary one or the other control point to make some adjustments, and within a few minutes you have designed the curve you want. For this purpose, it is convenient to have a program that allows both to watch the Bézier curve and to change the control points. We wrote a simple sample file that will demonstrate how to do this in Open Geometry("bezier_manip.cpp").

Example 2.35. A manipulator for Bézier curves
At first, we define an ordinary planar Bézier curve c. We draw the curve and its control polygon and mark the control points. For the manipulation we use two global variables:

Listing from `"bezier_manip.cpp"`:

```
int Index = 0;        // index of control point that is to be changed
Real Increment = 0.1; // increment for changing
```

Index gives the number of the active Bézier point, i.e., the Bézier point that you want to change. You will be able to translate it in the positive or negative direction parallel to the x- or y-axis. The translation vector will be of length Increment. Both variables can change their values during the animation. In Draw( ) we highlight the active control point in red color...

Listing from `"bezier_manip.cpp"`:

```
P2d P = BezierCurve.GetControlPoint( Index );
P.Mark( Red, 0.3, 0.2 );
```

...before we listen to the user's keystroke:

Listing from `"bezier_manip.cpp"`:

```
int key = TheKeyPressed(), n = BezierCurve.GetPNum( );
switch (key )
{
case 'u':
   P.y += Increment;
   BezierCurve.SetControlPoint( Index, P );
   break;
case 'd':
   P.y -= Increment;
   BezierCurve.SetControlPoint( Index, P );
   break;
case 'l':
   P.x -= Increment;
   BezierCurve.SetControlPoint( Index, P );
   break;
case 'r':
   P.x += Increment;
```

```
    BezierCurve.SetControlPoint( Index, P );
    break;
  case '+':
    Index++;
    Index %= n;
    break;
  case '-':
    Index--;
    if ( Index == -1 )
      Index = n-1;
    break;
  case 'i':
    Increment *= 0.5;
    break;
  case 'I':
    Increment *= 2;
    break;
  case 'e':
    BezierCurve.ElevateDegree( 1 );
    break;
  case 'p':
    PrintData( );
    break;
  }
```

It is intended that the user start the animation by pressing <Ctrl+F> or clicking on the Fps button in the button bar. Then the computer will check whether a new user input has arrived with every new frame and — according to the respective key — perform a certain action. These actions concern the translation of the active control point, the changing of Index and Increment, the degree elevation of the Bézier curve, and the output of the control polygon.

The last point is important because you probably want to use the Bézier curve you have designed for another purpose. We wrote a little routine for that purpose:

Listing from "bezier_manip.cpp":

```
void PrintData( )
{
  int i, n = BezierCurve.GetPNum( );
  for ( i = 0; i < n; i++ )
  {
    BezierCurve.GetControlPoint( i ).Print( );
  }
}
```

It will write the necessary data to OPEN GEOMETRY's standard output file "try.log". If you view this file with an arbitrary text editor, you will see something like

```
( -10.000, -7.000, 0.000 )
( -8.000, 4.000, 0.000 )
( 5.000, 7.000, 0.000 )
( 10.000, -7.000, 0.000 )
```

In order to use this data in another OPEN GEOMETRY program, you have to edit this file and adapt it to the syntax of C++. Of course, you can use it with other programs as well. The text output in "try.log" is rather simple. If you want to use the control polygon data frequently, you should probably write your own output routine, either with the help of standard C output commands or with OPEN GEOMETRY's PrintString(...) (compare page 561).

The program can easily be adapted to Bézier curves in space and to rational Bézier curves. You will need a few additional command keys, but the basic principles remain the same. In any case, you should not forget to print an instruction on the screen. In "bezier_manip.cpp" the corresponding code lines read as follows:

Listing from "bezier_manip.cpp":

```
const Real x = 10.4, y = 0;

PrintString( Black, x, y + 8.5, "Start Animation (Ctrl +'f')," );
PrintString( Black, x, y + 8,
       "then manipulate the control points:" );

PrintString( Green, x + 6, y + 6, "'u'...move up" );
PrintString( Green, x + 6, y + 5.3, "'d'...move down" );
PrintString( Green, x + 6, y + 4.6, "'l'...move left" );
PrintString( Green, x + 6, y + 3.9, "'r'...move right" );

PrintString( Green, x + 6, y + 2.2, "'+'...index++" );
PrintString( Green, x + 6, y + 1.5, "'-'...index--" );
PrintString( Green, x + 6, y + 0.8, "'l'...double increment" );
PrintString( Green, x + 6, y + 0.1, "'i'...half increment" );
PrintString( Green, x + 6, y - 0.6, "'e'...elevate degree" );
PrintString( Green, x + 6, y - 1.3, "'p'...print data" );

PrintString( Red, x, y + 6, "index =%2.0f", Index );
PrintString( Red, x, y + 5.3, "increment =%2.2f", Increment );
PrintString( Red, x, y + 4.6, "P.x =%2.2f", P.x );
PrintString( Red, x, y + 3.9, "P.y =%2.2f", P.y );
PrintString( Red, x, y + 3.2, "P.z =%2.2f", P.z );
```

We display a list of command keys and show the current value of the important variables. Additionally, we draw the coordinate axes and a coordinate grid in order to give some orientation on the screen. ♢

Example 2.36. Edge-orthogonal Bézier patches
As another example of the use of the class *BezierCurve2d*, we present the solution to a problem from the field of technical engineering. In the grid design for numerical evaluation of supersonic turbines, special pairs of plane Bézier curves are needed ([10]).

Given an arbitrary Bézier curve $b\colon B(u)\ldots\vec{b}(u)$ of order N (in our program `"edge_orthogonal.cpp"` it will be called BaseCurve), we are looking for a Bézier curve $e\colon E(u)\ldots\vec{e}(u)$ of order N with the property that the normal $n(u)$ of b in $B(u)$ is the straight line through $B(u)$ and $E(u)$. One can imagine b and E as borderlines of a plane Bézier patch $\Phi\colon X(u,v)\ldots\vec{x}(u,v)$ of order $(N,1)$ where the v-parameter lines intersect b orthogonally. Thus, one says that b and e determine an *edge-orthogonal Bézier patch* (compare [22] and Figure 2.56).

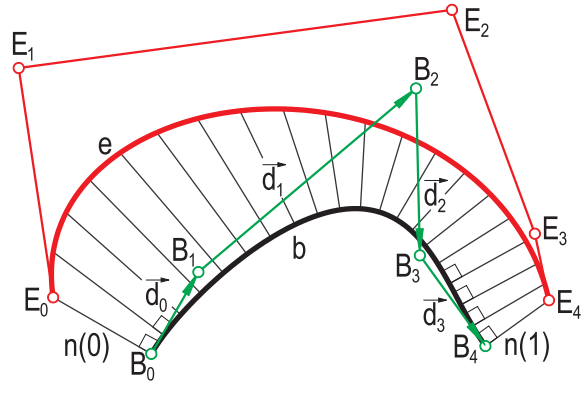

FIGURE 2.56. Two Bézier curves b and e that determine an edge-orthogonal Bézier patch.

In `"edge_orthogonal.cpp"` we used two completely different approaches to solve the problem. In general, there exists a two-parametric variety of exact solutions. It is clear that for any exact solution the control points E_0 and E_{N-1} of e must lie on the normals $n(0)$ and $n(1)$ of b through start and end point, respectively. Reversed, any pair (E_0, E_{N-1}) of points on $n(0)$ and $n(1)$ uniquely determines an exact solution. Explicit formulas for the control points are known, and it is no problem to display the result in Open Geometry. We will return to this a little later.

The second approach is completely different. We are not looking for an exact solution but for a solution of *sufficient accuracy for practical use*. In our simple case this has no advantages compared to the exact solution, but we can generalize

the concept in different ways: We may look for edge-orthogonal patches in higher dimensions or for edge-orthogonal patches with a rational control structure (a weight is assigned to each control point). Furthermore, we will no longer be restricted to right angles and can ask for an arbitrary angle function along the base curve. Even some fantasy conditions on the patch may (almost) be fulfilled, despite the absence of an exact solution!

The basic idea is simple: Given the base curve b, we start with an arbitrary test curve t: $T(u)\ldots\vec{t}(u)$. If $\alpha(u)$ denotes the angle between $B(u)T(u)$ and the normal of b in $B(u)$, we may use the *score*

$$s := \int_0^1 |\alpha(u)|\, du \tag{7}$$

to judge the quality of t. This score is, of course, not the only possibility. We may, for example, optimize the integral over $|\alpha(u)|^2$ or the maximal angle deviation. In any case, a small score indicates good approximations to an edge-orthogonal patch. A score equal to 0 characterizes the exact solutions.

Now we replace a control point T_i of t by a random point T^* and compute the new score s^*. If $s^* < s$, we accept the change; otherwise, we refuse it. Thus, we get a sequence $\langle t_i \rangle$ of test curves that induces a monotonical decreasing (and thus convergent!) sequence $\langle s_i \rangle$ of scores. In the end we can hope to get a satisfying result. The whole procedure is an application of the well-known *Monte Carlo method*.

Before implementing this idea in Open Geometry, we have to decide on certain details of a reasonable strategy. In `"edge_orthogonal.cpp"` we rely on the following:

- We create a random vector $\vec{v}$ of a constant length (e.g., 0.5 units).
- We replace the control point $T_{n(i)}$ of t by $T^*_{n(i)}\ldots\vec{t}_{n(i)} + \vec{v}$ and accept or reject the change according to the criterion described above.
- The sequence $\langle n_i \rangle$ takes the values $1, 2, \ldots, N-2, N-1, N-2, \ldots, 1, 0$ and continues periodically.[24]

Now to the corresponding code. In Init( ) we define three Bézier curves b, t (TestCurve), and e (ExactSolution).

[24]This "forward and backward" process takes into account the formal symmetry of Bézier curves: Reversing the sequence of the control points does not change the curve's shape.

Listing from "edge_orthogonal.cpp":

```
void Scene::Init( )
{
   P2d B[N];
   B[0]( -10, -6 );
   B[1]( -7, -7.5 );
   B[2]( -4, -8 );
   B[3]( 0, -8 );
   B[4]( 2, -6 );
   BaseCurve.Def( Green, 200, N, B );

   P2d T[N];
   int i;
   for ( i = 1; i < N1; i++ )
      T[i].Def( rnd_x( ), rnd_y( ) );
   T[0] = B[0] + 3 * fabs( test_x( ) )
            * BaseCurve.NormalVector( 0 );
   T[N-1] = B[N1] + 3 * fabs( test_x( ) )
            * BaseCurve.NormalVector( 1 );
   TestCurve.Def( Red, 200, N, T );

   V2d v[N];
   Real alpha = GetAlpha( &BaseCurve, &TestCurve );
   Real beta = GetBeta( &BaseCurve, &TestCurve );
   P2d E[N];
   for ( i = 1; i < N1; i++ )
   {
      v[i] = alpha * i / N1 * ( B[i] - B[i-1] ) +
           beta * ( N1 - i ) / N1 * ( B[i+1] - B[i] );
      v[i] = V2d( -v[i].y, v[i].x );
      E[i] = B[i] + v[i];
   }
   E[0] = TestCurve.GetControlPoint( 0 );
   E[N1] = TestCurve.GetControlPoint( N1 );
   ExactSolution.Def( Blue, 200, N, E );
}
```

The global constant N = 5 is the order of our curves, b is defined in the usual way, while the control points of t are random points from a certain area.[25] The first and the last control point of t are restricted to the normals $n(0)$ and $n(1)$ of b.

In the last third of Init( ) we compute the exact solution. First, the mathematical formulas: We denote the coordinate vectors of the control points of b by $\vec{b}_i$. Then the coordinate vectors of the control points $\vec{e}_i$ of e can be computed according to

$$\vec{e}_i = \vec{b}_i + \alpha\frac{i}{N}\mathbf{D}\vec{d}_{i-1} + \beta\frac{N-i}{N}\mathbf{D}\vec{d}_i, \tag{8}$$

where

$$\alpha, \beta \in \mathbb{R}, \quad \vec{d}_i := \vec{b}_{i+1} - \vec{b}_i, \quad \text{and} \quad \mathbf{D} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}.$$

In our example we draw the exact solution curve through the start and end points of the test curve. This allows a comparison of test curve and exact solution. You will see that they may differ considerably even though score and maximal angle deviation are very small. We have to compute the corresponding reals α and β first. Since we will need this later, we write our own functions for this task:

Listing from "edge_orthogonal.cpp":

```
Real GetAlpha( BezierCurve2d *c, BezierCurve2d *d )
{
    int m = c->GetPNum( );
    m--;
    int n = d->GetPNum( );
    return c->GetControlPoint( m ).Distance
           (d->GetControlPoint( n - 1 ) ) /
           c->GetControlPoint( m ).Distance
           ( c->GetControlPoint( m - 1 ) );
}
```

The input parameters of GetAlpha( ) are two Bézier curves, in our example the base curve and the test curve. It returns the value $\overline{B_N T_N}/\overline{B_N B_{N-1}}$. The analogous function GetBeta( ) returns $\overline{B_0 T_0}/\overline{B_0 B_1}$, and we can use both to define the exact solution according to (8).

Now to the approximation process. In Animate( ) we determine the index of the point to be changed. We initialize a global variable Index of type integer with 1 and change it in Animate( ) according to the following code:

[25] Remember to use *global* variables Rnd_x( ) and Rnd_y( ) of type *RandNum*! If Rnd_x( ) and Rnd_y( ) are local, they will not change their value during a fast animation.

Listing from "edge_orthogonal.cpp":

```
void Scene::Animate( )
{
  if ( ( FrameNum( ) ) % ( N2 ) < N1 )
     Index += 1;
  else
     Index -= 1;
}
```

Here N1 = N−1 and N2 = 2N−2. This yields the periodic sequence we mentioned above. In Draw( ) we use two auxiliary functions. The first returns the angle between the normal $n(u)$ of b and the straight line $B(u)T(u)$:

Listing from "edge_orthogonal.cpp":

```
Real Angle( BezierCurve2d *c, BezierCurve2d *d, Real u )
{
   return c->NormalVector( u ).Angle( V2d( c->CurvePoint( u ),
                          d->CurvePoint( u ) ), false, true );
}
```

The second computes the score according to (7). Unfortunately, we cannot use OPEN GEOMETRY's Integral(...) function, since we need pointers to Bézier curves as additional input parameters. It is, however, quite easy to adapt the source code of Integral(...) from "h.cpp" to our needs.

Listing from "edge_orthogonal.cpp":

```
Real CalcScore( BezierCurve2d (*c), BezierCurve2d (*d), int n )
{
   if ( n % 2 ) n++;
   Real area = fabs( Angle( c, d, 0 ) ) + fabs( Angle( c, d, 1 ) );
   Real dt = (Real) 1 / n;
   Real t = dt;
   int i, m = 4;
   for ( i = 1; i < n; i++ )
   {
      area += m * fabs( Angle( c, d, t ) );
      m = 6 - m;
      t += dt;
   }
   return area * dt / 3;
}
```

The input parameters are two pointers to Bézier curves and an integer value n that determines the number of intervals to be taken for the approximation of the integral through Simpson's formula. Now we take our chances by changing the right control point of the test curve:

Listing from "edge_orthogonal.cpp":

```
P_old = TestCurve.GetControlPoint( Index );
if ( ! (Index == 0) && ! (Index == N1) )
{
    V2d v( Test_x( ), Test_y( ) );
    v.Normalize( );
    v *= VectorLength;
    P = P_old + v;
}
else if ( Index == 0 )
{
    V2d v = BaseCurve.NormalVector( 0 );
    v *= Test_x( );
    v *= VectorLength;
    P = P_old + v;
}
else if ( Index == N1 )
{
    V2d v = BaseCurve.NormalVector( 1 );
    v *= Test_x( );
    v *= VectorLength;
    P = P_old + v;
}
TestCurve.SetControlPoint( Index, P );
```

The variables P and P_old are global variables of type *P3d*. We store the old control point in P_old and create a new random point P simply by adding the random vector v. If the first or last point of the control polygon is to be changed, we restrict the change to the normals $n(0)$ and $n(1)$ of b. In a next step, we compute the score and accept the change if it has improved. Otherwise, we reject it.

Listing from "edge_orthogonal.cpp":

```
Score = CalcScore( &BaseCurve, &TestCurve, 100 );
int i;
if( Score_old < Score )
{
    TestCurve.SetControlPoint( Index, P_old );
    PrintString( Black, 7, 3.5, "Score =%2.6f", Score_old );
}
else
{
    PrintString( Black, 7, 3.5, "Score =%2.6f", Score );
    Score_old = Score;

    if( (Index == 0) || (Index == N1) )
    {
        Real alpha = GetAlpha( &BaseCurve, &TestCurve );
        Real beta = GetBeta( &BaseCurve, &TestCurve );
        EdgeOrthogonalPatch( alpha, beta,
                             &BaseCurve, &ExactSolution );
    }
}
```

In addition, we print the current score on the screen and recompute the exact solution if T_0 or T_N has changed. For this last task we wrote a function of our own (EdgeOrthogonalPatch(...)) that more or less reads like the code we used in Init( ) to initialize e. The main difference is that we have to employ pointers and dynamic memory allocation, since we use two input Bézier curves.

Now we implement some additional features in order to be able to judge the quality of the solution. We compute the maximal angle deviation and draw the angle function. Furthermore, we draw a distance curve to b. The distance function we use is proportional to the angle error. Thus, you can easily identify the regions of good and not so good approximation on b.

Listing from "edge_orthogonal.cpp":

```
const int number_of_points = 300;
Real u;
Real delta = (Real) 1 / ( number_of_points - 1 );
Real max_deviation = 0;
Real deg = 0;
for ( i = 0, u = 0; i < number_of_points; i++, u+= delta )
{
   deg = Angle( &BaseCurve, &TestCurve, u );
   max_deviation = max( max_deviation, fabs( Deg( deg ) ) );
```

```
    P2d F( 4 * u + 7, Angle( &BaseCurve, &TestCurve, u ) + 6 );
    F.Mark( Black, 0.04 );
    P2d E = BaseCurve.CurvePoint( u ) +
      Deg( deg ) * ErrorFactor * BaseCurve.NormalVector( u );
    E.Mark( Yellow, 0.04 );
  }
  StraightLine2d( Black, P2d( 7, 6 ), P2d( 11, 6 ), THIN );
```

Finally, we imitate a plane Bézier patch by drawing the connecting lines of corresponding points $B(u)$ and $T(u)$ on b and t. We draw the Bézier curves and their control polygons and print the information of interest on the screen. A last good thing is the possibility of changing the length of the random vector. Otherwise, no change would occur after a while, since any random attempt yields a worse score. We implement this as follows:

Listing from `"edge_orthogonal.cpp"`:

```
int key = TheKeyPressed();
switch (key)
{
case 'd':
  VectorLength *= 2;
  break;
case 'f':
  VectorLength *= 0.5;
  break;
}
```

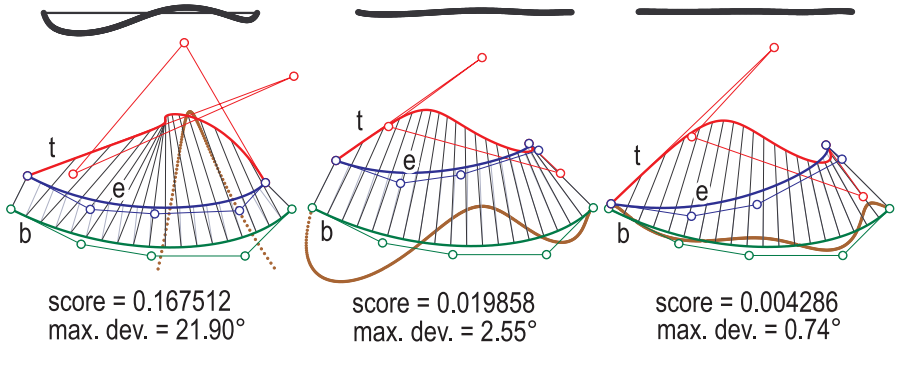

FIGURE 2.57. Three steps on the way to finding an edge-orthogonal Bézier patch. The test curve at the start (left), after 500 frames (middle), and after 1700 frames (right). At the end there is no visible deviation from the right angle condition.

We can thus interactively control the length of the random vector. Figure 2.57 shows three steps on the way to an almost exact solution. ♢

Example 2.37. Bézier approximation
We have mentioned earlier that the Monte Carlo method of the previous example can be generalized to many other ideas. So, why not try to approximate arbitrary parameterized curves through Bézier curves? We did this in `"bezier_approx.cpp"`.

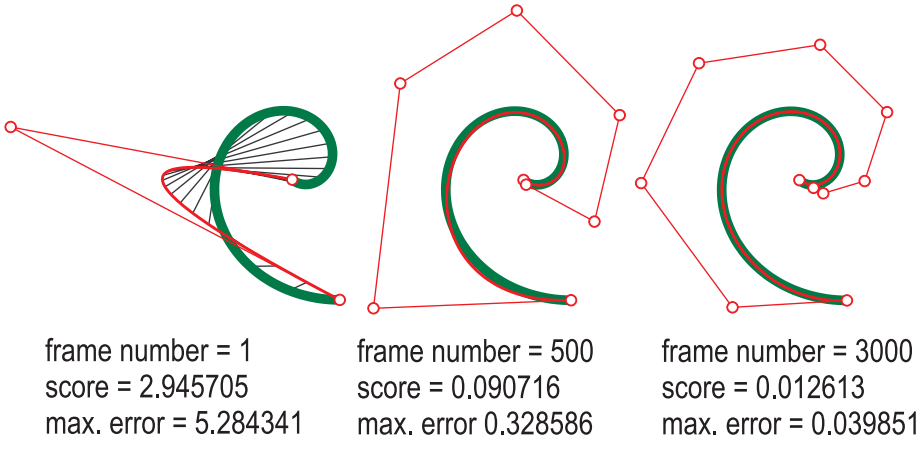

FIGURE 2.58. The test curve converges to the target curve (equiangular spiral). After 3000 frames the maximal error is almost invisible.

First, we define the *target curve* t: $T(u)\ldots\vec{t}(u)$ (i.e., the curve we want to approximate). It is an object of type *ParamCurve2d* with one special property.

Listing from `"bezier_approx.cpp"`:

```
class MyCurve: public ParamCurve2d
{
public:
    P2d CurvePoint( Real u ) const
    {
        Real u0 = -2, u1 = 5;
        u = ( 1 - u ) * u0 + u * u1;
        Real r = exp( 0.3 * u );
        return P2d( r * cos( u ), r * sin( u ) );
    }
};
MyCurve TargetCurve;
```

We specify the parameter interval $[\mathsf{u0}, \mathsf{u1}]$ of t in CurvePoint(...) and transform it immediately to $[0, 1]$. Hence, we can define the target curve in Init( ):

Listing from "bezier_approx.cpp":

```
TargetCurve.Def( Green, 200, 0, 1 );
```

This is necessary, since we want to approximate a parameterized curve through a Bézier curve $b\colon B(u) \ldots \vec{b}(u)$ (i.e., we want to have $B(u) \approx T(u)$) and the standard parameter interval for Bézier curves is $[0, 1]$. In our case, the target curve is an equiangular spiral, a hard task, since the curvature changes rather fast along the curve. But you will see that we are able to get satisfying results.

The quality of a solution will be judged by the score function

$$\int_0^1 \operatorname{dist}(B(u)T(u))\, dt.$$

We implement it in more or less the same way as in "edge_orthogonal.cpp":

Listing from "bezier_approx.cpp":

```
Real DistError( ParamCurve2d *target, BezierCurve2d *test,
    Real u )
{
    return target->CurvePoint( u ).Distance(
          test->CurvePoint( u ) );
}
Real CalcScore( ParamCurve2d *target, BezierCurve2d *test, int n )
{
    if ( n % 2 ) n++;
    Real area = DistError( target, test, 0 ) +
                DistError( target, test, 1 );
    Real dt = (Real) 1 / n;
    Real t = dt;
    int i, m = 4;
    for ( i = 1; i < n; i++ )
    {
        area += m * DistError( target, test, t );
        m = 6 - m;
        t += dt;
    }
    return area * dt / 3;
}
```

Next, we define the first test curve in Init( ). The first and the last points will be the start and end points of the target curve t. The initial degree of t is just 3. We will be able to elevate it interactively during the program.

Listing from "bezier_approx.cpp":

```
P2d T[3];
T[0] = TargetCurve.CurvePoint( 0 );
T[1].Def( rnd_x( ), rnd_y( ) );
T[2] = TargetCurve.CurvePoint( 1 );
TestCurve.Def( Red, 200, 3, T );
```

The next steps are analogous to those in "edge_orthogonal.cpp". We apply a random change to a control point of the test curve, compute the new score, and accept the change only if the score has improved. A little difference is the sequence of indices of control points. We do not need to change the first and last points, so this sequence takes the values $\langle 1, 2, \ldots, n-2, n-1, n-1, \ldots, 2, 1 \ldots \rangle$ and continues periodically. We implement it in Animate( ) as follows:

Listing from "bezier_approx.cpp":

```
Index = FrameNum( ) % ( N2 );
if ( Index > N3 )
    Index = N2 - Index;
else
    Index++;
```

A very important feature is the possibility of elevating the degree of the test curve during the random process. We can do this in the same way as we control the length of the random vector. At the end of Draw( ) we write the following lines:

Listing from "bezier_approx.cpp":

```
int key = TheKeyPressed();
switch (key )
{
case 'd':
    VectorLength *= 2;
    break;
case 'f':
    VectorLength *= 0.5;
    break;
case 's':
    ElevateDeg = true;
}
```

ElevateDeg is a global variable of type *Boolean*. If it is true, we elevate the degree of the test curve and adapt two global variables in Animate( ):

Listing from `"bezier_approx.cpp"`:

```
if ( ElevateDeg )
{
   TestCurve.ElevateDegree( 1 );
   N++;
   N2 = 2 * N - 4;
   N3 = N - 3;
   ElevateDeg = false;
}
```

If you run the program, you will see the target curve and a random Bézier curve of degree 2 through start and end points of the target curve. Press <`Ctrl + F`> and use the keys **s**, **d**, and **f** for interactive control of the random process. We recommend the following strategy for fast convergence:

- Reduce the vector length until the score changes rather quickly.
- If the score does no longer improve considerably, increase the vector length.
- Elevate the degree of the test curve and start again by reducing the vector length step by step.

Figure 2.58 shows the convergence of the test curve. ♢

B-spline curves

So far, we have extensively dealt with Bézier curves and their useful properties for CAGD. However, they have two serious disadvantages that must not be forgotten:

1. If you want to display a complex shape, you need a very high number of control points.
2. The changing of a single control point or a single weight will affect the whole curve.

The common technique to overcome these difficulties is the use of *spline curves*: One combines different parts of integral or rational Bézier curves. An appropriate choice of the control polygons of adjacent curves guarantees GC^k-continuity or

C^k-continuity, i.e., the different curve segments will form a smooth curve. We have shown how to do this in Example 2.34.

The curve segments are the *spline segments*; the whole curve is called a *Bézier spline*. However, the most frequent spline technique does not use Bézier splines but *B-splines*. Theoretically, there is no essential difference between Bézier splines and B-splines: Both approaches yield the same class of curves. But the storage of the control data is more efficient with B-splines.

The theoretic description of B-spline curves is more complex than the description of Bézier curves.[26] Their *base functions* N_i^k are recursively defined via a *knot vector* $\mathbf{T} = (t_0, \ldots, t_{n-1}, t_n, t_{n+1}, \ldots, t_{n+k})$ with $t_i \in \mathbb{R}$ and $t_0 \leq t_1 \leq \cdots \leq t_{n+k}$. We start by setting

$$N_i^1(t) := \begin{cases} 1 & \text{for } t_i \leq t < t_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

and define recursively for $k > 1$,

$$N_i^k(t) = \frac{t - t_i}{t_{i+k-1} - t_i} N_i^{k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1}^{k-1}(t). \tag{10}$$

The B-spline base functions have the following properties (compare [18] and Figure 2.59):

1. $N_i^k(t) > 0$ for $t_i < t < t_{i+k}$.
2. $N_i^k(t) = 0$ for $t_0 \leq t \leq t_i$ and $t_{i+k} \leq t \leq t_{n+k}$.
3. $\sum_{i=0}^{n} N_i^k(t) = 1$ for $t \in [t_{k-1}, t_{n+1}]$.
4. For $t_i \leq t_l \leq t_{i+k}$ the base functions $N_i k(t)$ are C^{k-2}-continuous at the knots t_l.

Points 1 and 2 guarantee that the influence of each base function is limited to a certain well-defined *supporting interval*. Point 3 is responsible for the affine invariance of curve and control polygon, and point 4 allows one to control the degree of continuity via the integer k.

Note that coinciding knots $t_i = t_{i+1} = \cdots = t_{i+j}$ are allowed. In fact, some basic B-spline techniques rely on these *multiple* knots.

[26] It would be a miracle if this were not the case. We have to describe a sequence of spline segments with a control structure and additional information about the degree of continuity, not just a single curve.

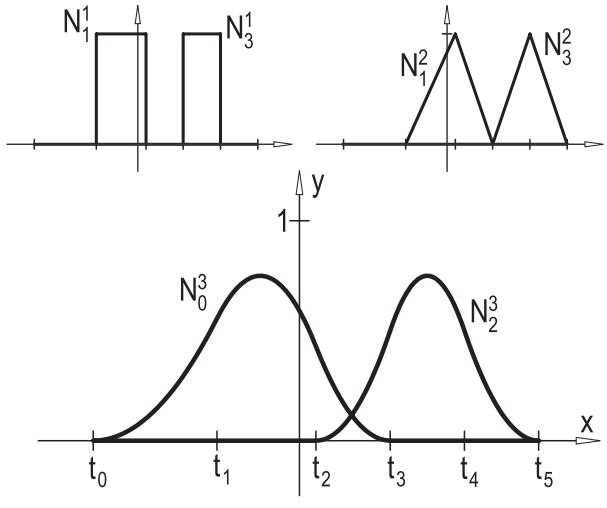

FIGURE 2.59. Six B-spline functions over the knot vector $\mathbf{T} = (t_0, \ldots, t_5)$.

Example 2.38. B-spline base functions
It is a nice and interesting task to write a program to display the B-spline base function (`"base_functions.cpp"`): We will use a recursive function call, and overwriting some methods, we derive a successor class of *ParamCurve2d*. At first, we declare global constants for the knot vector T, the colors, and the scale factors in the x and y directions.[27]

Listing from `"base_functions.cpp"`:

```
const int M = 8;
const Real T[M] = { -4, -2.5, -1, 0.2, 1.1, 2, 2.9, 4 };
const int K = 8; // maximal order of the B-spline base functions
const Color Col[10] = { Brown, Red, Yellow, Green, Blue, Orange,
                        Cyan, Pink, Magenta, LightYellow };
const Real ScaleX = 3, ScaleY = 8;
```

Then we define a recursive function SplineN(...) according to equation (10):

[27] The B-spline base functions are too small to fit in a standard Open Geometry window.

Listing from "base_functions.cpp":

```
Real SplineN( Real t, int i, int k )
{
   if ( k == 1 )
      return ( ( t <= T[i] ) || ( T[i+1] < t ) ) ? 0 : 1;
   else
   {
      return ( t - T[i] ) / ( T[i+k-1] - T[i] ) *
         SplineN( t, i, k - 1 ) +
         ( T[i+k] - t ) / ( T[i+k] - T[i+1] ) *
         SplineN( t, i + 1, k - 1 );
   }
}
```

With the help of this family of functions we define the base functions as parameterized curves. We use two private member variables and overwrite the defining method and drawing method (taking into account the respective supporting interval). If $K = 1$ or $K = 2$, the B-spline base functions consist of line segments, and we draw them directly.

Listing from "base_functions.cpp":

```
class BSplineBase: public ParamCurve2d { public:
   void Def( Color c, int PointNum, int i, int k )
   {
      I = i, K = k;
      ParamCurve2d::Def( c, PointNum,
         min( 0, T[I] ), min( T[M-1], T[I+K] ) );
   }
   virtual P2d CurvePoint( Real u )
   {
      return P2d( ScaleX * u, ScaleY * SplineN( u, I, K ) );
   }
   void Draw( ThinOrThick style, Real max_dist = -1 )
   {
      if ( I > 0 )
         StraightLine2d( col, P2d( ScaleX * T[0], 0 ),
            P2d( ScaleX * T[I], 0 ), style );
      if ( I + K < M - 1 )
         StraightLine2d( col, P2d( ScaleX * T[I+K], 0 ),
         P2d( ScaleX * T[M-1], 0 ), style );
      if ( K == 1 && I < M - 1 )
      {
         StraightLine2d( col, P2d( ScaleX * T[I], 0 ),
```

```
            P2d( ScaleX * T[I], ScaleY ), style );
        StraightLine2d( col, P2d( ScaleX * T[I], ScaleY ),
            P2d( ScaleX * T[I+1], ScaleY ), style );
        StraightLine2d( col, P2d( ScaleX * T[I+1], ScaleY ),
            P2d( ScaleX * T[I+1], 0 ), style );
    }
    else if ( K == 2 && I < M - 2 )
    {
        StraightLine2d( col, P2d( ScaleX * T[I], 0 ),
            P2d( ScaleX * T[I+1], ScaleY ), style );
        StraightLine2d( col, P2d( ScaleX * T[I+1], ScaleY ),
            P2d( ScaleX * T[I+2], 0 ), style );
    }
    else
        ParamCurve2d::Draw( style, max_dist );
  }
private:
  int I, K;
};
```

Next, we declare a 2D array of instances of class BaseF and define it in Init( ):

Listing from `"base_functions.cpp"`:

```
BSplineBase BaseF[M][K];

void Scene::Init( ) {
  int i, k;
  for ( k = 1; k < K; k++ )
    for ( i = 0; i < M - k; i++ )
      BaseF[i][k].Def( Col[k], 61, i, k );
}
```

In Draw( ) we repeat this loop with a draw command and display the coordinate axes plus a scale indicating the knots (on the x-axis) and the unit (on the y-axis). With small adaptations this program was used to produce Figure 2.59. ♢

Now we proceed with the introduction to B-spline curves. A B-spline is a piecewise polynomial curve whose segments are of continuity class C^κ ($\kappa \geq 0$). We choose a knot vector $\mathbf{T} = (t_0, \ldots, t_n, t_{n+1}, \ldots, t_{n+k})$ and a sequence of control points $\vec{d}_0, \ldots, \vec{d}_n$ in $\mathbb{R}^2$ or $\mathbb{R}^3$. A *B-spline curve b of order k* with control points $\vec{d}_i$ and knot vector $\mathbf{T}$ is defined as

$$b \ldots \vec{x}(t) = \sum_{i=0}^{n} \vec{d}_i N_i^k(t). \tag{11}$$

The parameter interval is usually restricted to $[t_{k-1}, t_{n+1}]$ because only there is the full range of B-spline base functions available. Of course, this does not mean that the knots $t_0, \ldots, t_{k-2}$ and $t_{n+2}, \ldots, t_{n+k}$ have no effect on the shape of the B-spline. Chosen in an appropriate way, they can guarantee certain nice properties of the curve (we shall see examples of this later in this chapter).

In general, a B-spline curve is of C^{k-2}-continuity: The $(k-2)$-nd derivative $\vec{x}^{(k-2)}$ is still continuous, while $\vec{x}^{(k-1)}$ is not. If, however, a knot is of multiplicity l (i.e., it occurs l-times in the knot vector $\mathbf{T}$), the continuity class reduces to C^{k-l-1}.

In Figure 2.60 (left-hand side) we display a C^2-continuous B-spline curve with six control points $\vec{d}_i$ and a uniform knot vector $\mathbf{T} = (0, 1, \ldots, 9)$ (i.e., $n = 5$ and $k = 4$). The connection between curve and control polygon is not as obvious as in the case of Bézier curves.

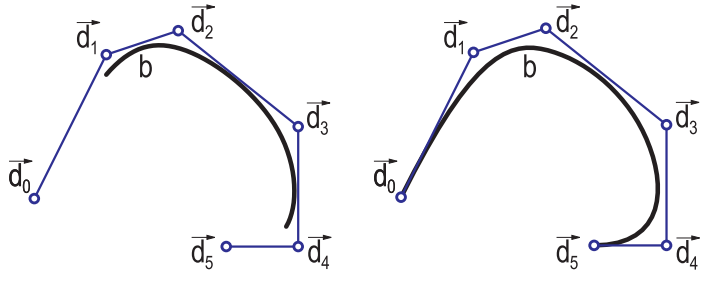

FIGURE 2.60. Two B-spline curves with the same control polygon but different knot vectors: $\mathbf{T} = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$ for the left and $\mathbf{T} = (0, 0, 0, 0, 1, 2, 3, 3, 3, 3)$ for the right curve.

This changes if we consider the B-spline curve with the same control polygon but knot vector $\mathbf{T} = (0, 0, 0, 0, 1, 2, 3, 3, 3, 3)$ (right-hand side of Figure 2.60). The first and the last knot have multiplicity k, which results in a behavior we already know from Bézier curves: $\vec{d}_0$ and $\vec{d}_5$ are start and end points of the B-spline, the first and the last side of the control polygon are curve tangents in these points.

These properties are key features for any application of B-splines in CAGD, and it will be useful to have a defining method that already cares for the correct choice of the knot vector. But first, we must have a look at the implementation of B-splines in Open Geometry 2.0.

We provide four classes of B-spline curves: *NUBS2d*, *NUBS3d*, *NURBS2d*, and *NURBS3d*. "NUBS" stands for "Non Uniformal B-Spline" and means that the knot vector need not consist of evenly distributed elements. "NURBS" means "Non Uniformal Rational B-Spline." It is defined in analogy to ordinary (integral) B-splines. However, as with rational Bézier curves, we can assign weights to the control points.[28]

In the following we will describe only the class *NUBS2d*. The remaining classes have been implemented in a similar way.[29]

NUBS2d is to successor of *ParamCurve2d*. Like all other B-spline classes, it is defined in `"nurbs.h"`. The header reads as follows:

Listing from `"H/nurbs.h"`:

```
class NUBS2d: public ParamCurve2d
{
public:
    NUBS2d( ) { T = A = NULL; D = E = NULL; }
    void Def( Color c, int size, int knum, Real K[], int pnum,
        P2d P[] );
    void Def( Color c, int size, int pnum, P2d P[], int continuity,
        Boolean closed = false );
    ~NUBS2d( ) { AllocMemory( 0, 0 ); }
    virtual P2d CurvePoint( Real u ) { return DeBoor( u ); }
    void MarkControlPoints( Color c, Real rad1, Real rad0 = 0 );
    void DrawControlPolygon( Color c, ThinOrThick style );
    Real GetKnot( int i ) {return T[i]; };
    P2d GetControlPoint( int i ) { return D[i]; };
    int GetKnotNum( ) { return M; }
    int GetPointNum( ) { return N; }
private:
    int n, k, N, M;
    Real *T, *A;
    P2d *D, *E;
    void AllocMemory( int knum, int pnum );
    P2d DeBoor( Real u ) const;
    int ComputeInterval( Real u ) const;
```

[28] This is completely analogous to the case of rational Bézier curves (compare page 130).

[29] We have prepared sample files that demonstrate the use of each of the mentioned classes (`"nubs2d.cpp"`, `"nubs3d.cpp"`, `"nurbs2d.cpp"`, `"nurbs3d.cpp"`).

```
    Boolean Closed;
};
```

The class provides a constructor and a destructor, two defining methods, and a few "getters." The CurvePoint(...) function is overwritten and calls the private member function DeBoor(...). The control points are stored in the *P2d* array D; the knots are stored in the *Real* array T. The arrays A and E are used for the computation of curve points in DeBoor(...).

The memory allocation for control points and knots must be dynamic. The corresponding OPEN GEOMETRY macros are hidden in AllocMemory(...). This method is also called by the destructor.

We could compute the curve points directly via equations (10) and (11). However, there exists a more efficient way of doing this: the subdivision algorithm of COX–DE BOOR. In [18] it is described as follows:

1. Given a B-spline curve b defined by a sequence $\langle \vec{d}_0, \ldots, \vec{d}_n \rangle$ of control points and its knot vector $\mathbf{T} = (t_0, \ldots, t_n, t_{n+1}, \ldots, t_{n+k})$ $(n \geq k-1)$, we want to compute the curve point corresponding to the parameter value $t^* \in [t_{k-1}, t_{n+1}]$.
2. Determine the index r with $t_r \leq t^* < t_{r+1}$; for $t^* = t_{n+1}$ use $r := n$.
3. For $j = r-k+2, \ldots, r$ compute
$$\alpha_j^1 := \frac{t^* - t_j}{t_{j+k-1} - t_j} \qquad \text{and} \qquad \vec{d}_j^{\,1} := (1-\alpha_j^1)\,\vec{d}_{j-1} + \alpha_j^1\,\vec{d}_j.$$
4. For $l = 2, \ldots, k-1$ and $j = r-k+l+1, \ldots, r$ compute
$$\alpha_j^l := \frac{t^* - t_j}{t_{j+k-l} - t_j} \qquad \text{and} \qquad \vec{d}_j^{\,l} := (1-\alpha_j^l)\,\vec{d}_{j-1}^{\,l-1} + \alpha_j^l\,\vec{d}_j^{\,l-1}.$$
5. The curve point in question is $\vec{d}_r^{\,k-1}$.

In the *NUBS2d* member function DeBoor(...) we follow this algorithm. With a minor adaptation we can avoid the allocation of 2D arrays for the reals α_i^j and the points $\vec{d}_i^{\,j}$. We use the one-dimensional arrays A and E for that purpose. Step 2 is done by calling the private member function ComputeInterval(...).

The preceeding listing shows two defining methods for a *NUBS2d* object. The first method leaves all responsibility to the user. The control points as well as the knots must be defined. The second method needs only the control points, an integer to define the class of differentiability, and a Boolean variable to decide whether the curve should be closed or not. This is a very convenient way, and unless you are absolutely sure of what you are doing, we recommend using the second method. We display a listing to show what happens:

Listing from `"H/nurbs.h"`:

```
void NUBS2d::Def( Color c, int size, int pnum, P2d P[],
    int continuity, Boolean closed )
{
    Closed = closed;
    if ( Closed )
    {
        N = pnum + continuity + 1, n = N - 1;
        k = min( continuity + 2, N );
        M = k + N;
        if ( N < k )
            SafeExit( "wrong index" );
        AllocMemory( M, N );
        int i;
        for ( i = 0; i < pnum; i++ )
            D[i] = P[i];
        int j;
        for ( i = pnum; i < N; i++ )
        {
            j = i % pnum;
            D[i] = P[j];
        }
        for ( i = 0; i < M; i++ )
            T[i] = i;
        ParamCurve2d::Def( c, size, T[k-1], T[n+1] );
    }
    else
    {
        N = pnum, n = N - 1;
        k = min( continuity + 2, N );
        M = k + N;
        if ( N < k )
            SafeExit( "wrong index" );
        AllocMemory( M, N );
        int i;
        for ( i = 0; i < N; i++ )
            D[i] = P[i];
        for ( i = 0; i < k; i++ )
            T[i] = 0;
        for ( i = n + 1; i < M; i++ )
            T[i] = n - k + 2;
        for ( i = k; i <= n; i++ )
            T[i] = i - k + 1;
        ParamCurve2d::Def( c, size, T[k-1], T[n+1] );
    }
}
```

Let's have a look at the **else**-branch first. There, the B-spline is not closed, and we already know what should happen: We want to produce a curve as on the right-hand side of Figure 2.60. We simply copy the control points and assign values to the knots so that $t_0 = \cdots = t_{k-1} = 0$ and $t_{n+1} = \cdots = t_{n+k} = n-k+2$ are k-fold knots.

The second option in this defining method sets closed to **true**. The output is a *closed* B-spline curve, which is quite useful for certain applications. In order to achieve this goal, we continue the sequence of control points periodically. The knot vector has no multiple knots.

If you want a special distribution of the knots, you can modify the described defining methods and adapt them to your needs. This is interesting if you deal with *interpolating* splines (so far we have talked only about *approximating* splines). Then it may, for example, be necessary to approximate the knot distance by the distance of the control points.

In Figure 2.61 we display a test series for open and closed B-spline curves with varying degree κ of continuity. In the case $\kappa = 0$, the "curve" is identical with its control polygon, if $\kappa = 1$, the sides of the control polygon are tangents of the B-spline. The curve in the upper right corner is a Bézier curve; it is even of differentiability class C^∞.

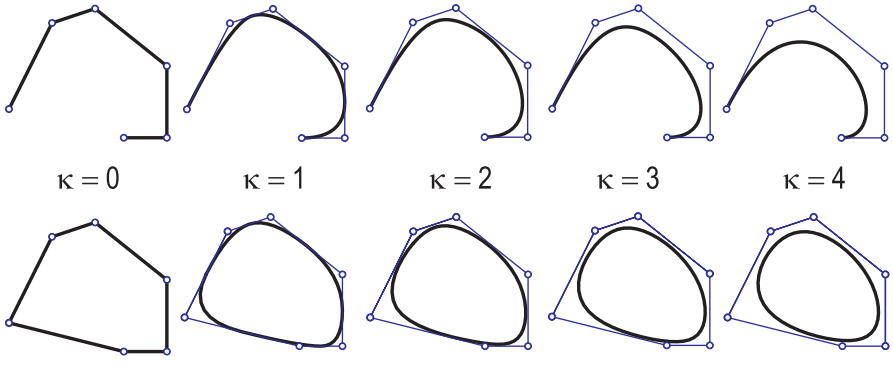

FIGURE 2.61. Different B-spline curves.

B-splines in 3D and rational B-splines work in more or less the same way. Sample code and description can be found in Chapter 6.

2.7 Further Examples

Example 2.39. Minimize distance
With a little imagination, it is easy to find many practical applications of the following abstract geometrical problem: Given a finite set $\mathcal{F} := \{F_0, \ldots, F_n\}$ of

points, we want to find a point P that lies "as close as possible" to all points F_i. The meaning of the phrase "as close as possible" needs, of course, a more precise explanation. It seems sensible to minimize either the sum of distances $\overline{PF_i}$ or the maximal distance. Perhaps, the barycenter B of $\mathcal{F}$ is a good choice as well. The best choice depends, of course, on the given practical problem. We will consider three approaches:

1. The distance sum $d(P) := \sum_{i=0}^{n} \overline{PF_i}$ is minimized.
2. The sum of squared distances $s(P) := \sum_{i=0}^{n} \overline{PF_i}$ is minimized.
3. The maximal distance $m(P) := \max\{\overline{PF_i} \mid i = 0, \ldots, n\}$ is minimized.

The functions $d(P)$, $s(P)$, and $m(P)$ will be called *score functions*. The solutions to the problem will be denoted by D (distance sum), S (squared distance sum), and M (maximal distance), respectively.[30] In general, these three approaches result in different solutions. The main difference is their behavior with respect to points that are far off from the main point cloud (Figure 2.62). Suppose, for example, that the points F_i are data points of a physical experiment. Then the exceptional point F_5 was probably just the result of a bad survey. One would prefer the solution D in this case. If one is not sure whether to take D or M one can decide on S. Its reaction to exceptional points lies somewhere between the two extreme cases D and M. By the way; S is always the barycenter of $\mathcal{F}$.

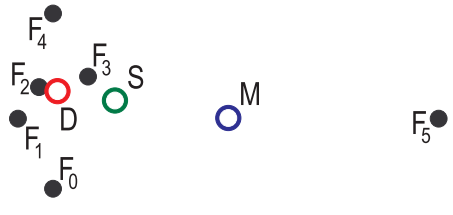

FIGURE 2.62. Three different approaches to the optimization problem. The distant point F_5 has different impact on the position of D, S, and M.

In our Open Geometry program `"minimize_distance.cpp"` we use a Monte Carlo method to determine D, S, and M (compare Examples 2.36 and 2.37!). We start with an arbitrary point $D = D_0$ and apply a small random translation to it. Then we compare the distance sums of the old and the new point. If it has improved, we accept the change; otherwise we refuse it. This results in a sequence $\langle D_0, D_1, \ldots \rangle$ and a strictly monotonically decreasing sequence $\langle d(D_i) \rangle$. Analogous procedures yield good approximations for S and M.

In `"minimize_distance.cpp"` we use the three functions to compute the values of the score functions $d(P)$, $s(P)$, and $m(P)$ (the points F_i are defined globally):

[30] It is quite easy to see that there always exists a solution, but it is not obvious that it is unique.

Listing from "minimize_distance.cpp":

```
Real DistSum( P2d &P )
{
    int i;
    Real sum;
    for ( i = 0, sum = 0; i < N; i++ )
        sum += P.Distance( F[i] );
    return sum;
}

Real SquareSum( P2d &P )
{
    int i;
    Real sum;
    for ( i = 0, sum = 0; i < N; i++ )
        sum += Sqr( P.Distance( F[i] ) );
    return sum;
}

Real MaxDist( P2d &P )
{
    int i;
    Real dist = 0;
    for ( i = 0; i < N; i++ )
        dist = max( dist, P.Distance( F[i] ) );
    return dist;
}
```

After initializing all points with sensible random values (of course, you can set the points F_i manually as well), we compute their scores and store them in ScoreD, ScoreS, and ScoreM, respectively. In Animate( ) we make analogous random attempts for D, S, and M. Here are the code lines for D:

Listing from "minimize_distance.cpp":

```
Real new_score;
V2d v;

P2d D_new = D;
v.Def( RndVX( ), RndVY( ) );
D_new += ChangeFactor * v;
new_score = DistSum( D_new );
if ( new_score < ScoreD )
{
    D = D_new;
    ScoreD = new_score;
}
```

After starting the program, the user has to press <Ctrl+F> to start the animation. The new random points will be computed and tested. The convergence of the point sequence is very fast at the beginning but soon slows down. The reason for this is that the random vectors are too large. That is why we use a change factor that determines the maximal length of the random vector. It can interactively be changed by pressing **f** or **d** during the animation. ♢

Example 2.40. The isoptic
Let c be an arbitrary conic and $\omega \in [0, 180]$ an arbitrary fixed angle. Then the *isoptic curve* $i(c, \omega)$ is defined as the locus of all points from which c is seen under an angle of ω. That is, the tangents of c through P form an angle of ω. One usually considers the isoptics $i(c, \omega)$ and $i(c, 180-\omega)$ as parts of one and the same curve, because they satisfy the same algebraic equation. It is problematic to derive a parameterized equation for the isoptic. But we can rely on a simple geometric construction (Figure 2.63):

We assume that c is an ellipse (an analogous construction is possible for the hyperbola). By F1 and F2 we denote its focal points, by a the half-length of the major axis. Let d be the circle with radius a centered at the ellipse's center. Now we take two straight lines through F2 that form an angle of ω. They intersect d in four points 1, 1′, 2, and 2′. The normals of F1 and F11′ through 1 and 1′ are tangents t_1 and t_1' of c. In the same way, we get tangents t_2 and t_2'. The vertices of the parallelogram $t_1 t_1' t_2 t_2'$ are four points of $i(c, \omega)$ or $i(c, 180 - \omega)$, respectively.

It is now immediately clear how to implement the isoptic in Open Geometry ("isoptic.cpp"). We do not compute anything; we simply perform the above construction:

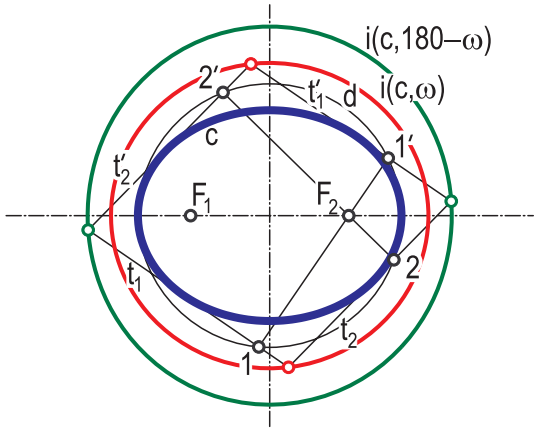

FIGURE 2.63. The construction of the isoptic curve of an ellipse.

Listing from `"isoptic.cpp"`:

```
class MyFirstIsoptic: public ParamCurve2d
{
public:
    P2d CurvePoint( Real u )
    {
        StrL2d s1, s2;
        s1.Def( F2, V2d( cos( u ), sin( u ) ) );
        s2 = s1;
        s2.Rotate( F2, Omega );
        P2d X1, X2;
        int n;
        n = Circle.SectionWithStraightLine( s1, X1, X2 );
        if ( s1.GetParameter( X1 ) > 0 )
            s1.Rotate( X1, 90 );
        else
            s1.Rotate( X2, 90 );
        n = Circle.SectionWithStraightLine( s2, X1, X2 );
        if ( s2.GetParameter( X1 ) > 0 )
            s2.Rotate( X1, 90 );
        else
            s2.Rotate( X2, 90 );
        return s1 * s2;
    }
};
MyFirstIsoptic Isoptic1;
```

When intersecting a straight line through F1 with the circle d, we do not need to check whether there exist real intersection points, since n will always take the value 2. We choose the intersection point on the positive half-ray to avoid discontinuities that stem from the order of the solution points. Note that this construction actually gives the points of $i(c, 180 - \omega)$. In order to get $i(c, \omega)$, we implement a second isoptic where we replace the first occurrence of ">" by "<". In the remaining parts we just define, draw, and mark relevant elements. We do not consider it necessary to display them here. ♢

Sometimes it is convenient or even necessary to regard a plane curve c as a set of straight lines (its tangents) rather than a set of points (its points of tangency). The principle of duality of projective geometry guarantees that both points of view are equivalent (in the sense of projective geometry), though the latter is much more frequent. The dual curve of a standard plane curve is called a *class curve*.

Example 2.41. Class curve
It is no problem to use a class curve in Open Geometry. Version 2.0 provides a successor class of *ParamCurve2d* that has all relevant methods:

Listing from `"H/lines2d.h"`:

```
class ClassCurve2d: public ParamCurve2d
{
public:
   virtual StrL2d Tangent( Real u ) = 0;
   virtual P2d CurvePoint( Real u )
   {
      return Tangent( u - 1e-6 ) * Tangent( u + 1e-6 );
   }
   virtual V2d TangentVector( Real u )
   {
      return Tangent( u ).GetDir( );
   }
};
```

We overwrite the Tangent(...), CurvePoint(...), and TangentVector(...) functions of *ParamCurve2d*. Note that *ClassCurve2d* is a *purely virtual function*. In order to use it we have to derive a successor and must implement the Tangent(...) function. We did this in `"class_curve1.cpp"` (R1 and R2 are positive reals):

Listing from "class_curve1.cpp":

```
class MyClassCurve: public ClassCurve2d
{
public:
   StrL2d Tangent( Real t )
   {
      P2d P( R1 * cos( t ), R1 * sin( t ) );
      P2d Q = P;
      Q *= R2 / R1;
      Q.Rotate( Origin2d, Phi );
      Q.Translate( Dir );
      return StrL2d( P, Q );
   }
};
MyClassCurve ClassCurve;
```

We connect two points P(t) and Q(t). The point P(t) lies on a circle c_1 centered at the origin with radius R1, Q(t) is derived from P(t) through a scaling, rotating, and translating operation. Therefore Q(t) lies on a circle c_2 of radius R2 and corresponds to P(t) in a homothety η that does not depend on t. Thus, we might say that our class curve c is generated by two homothetic circles. In Draw( ) we connect the corresponding points:

Listing from "class_curve1.cpp":

```
int i, n = 40;
Real t, delta = 2 * PI / n;
P2d P, Q;
for ( i = 0, t = 0; i < n; i++, t += delta )
{
   P.Def( R1 * cos( t ), R1 * sin( t ) );
   Q = P;
   Q *= R2 / R1;
   Q.Rotate( Origin2d, Phi );
   Q.Translate( Dir );
   StraightLine2d( Yellow, P, Q, THIN );
}
...
ClassCurve.Def( Blue, 150, -PI, PI );
ClassCurve.Draw( MEDIUM, 100 );
```

The class curve c itself is defined and displayed in Draw( ) as well. We have to define it in Draw( ) because we want to animate the image. In Animate( ) we will change the values of the rotation angle Phi and the translation vector Dir.

Listing from "class_curve1.cpp":

```
void Scene::Animate( )
{
   Phi += 1;
   Dir.Def( PulseX.Next( ), PulseY.Next( ) );
}
```

If you start the animation, you will immediately notice that the generated class curve c is always a conic section (Figure 2.64). It is well known that two projectively linked conics generate a curve of class four in general (correspondence principle of CHASLES). With each fixed point of the projectivity the class is reduced by one. In our case, the circular points at infinity remain fixed, and c is always of class two, i.e., a conic section.

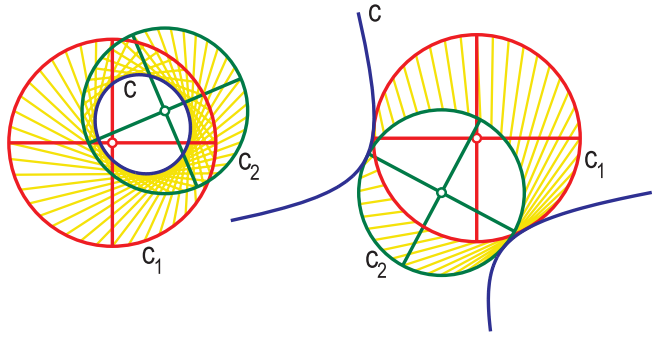

FIGURE 2.64. Two homothetic circles c_1 and c_2 generate a conic section c.

If you want to see a curve of class four, you must not use circles. In "class_curve2.cpp" we have replaced them by congruent ellipses e_1 and e_2. Using two global constants A and B and a global variable Y of type *Real*, the corresponding instance of *ClassCurve2d* is implemented as follows:

Listing from "class_curve2.cpp":

```
class MyClassCurve: public ClassCurve2d
{
public:
```

```
    StrL2d Tangent( Real u )
    {
        P2d P( A * cos( u ), B * sin( u ) );
        P2d Q( B * cos( u ), A * sin( u ) - Y );
        return StrL2d( P, Q );
    }
};
MyClassCurve ClassCurve;
```

Varying Y means translating the second ellipse e_2 along the y-axis. We use a pulsing real PulseY that swings harmonically between $A - B$ and $B - A$ to control this motion:

Listing from `"class_curve2.cpp"`:

```
void Scene::Draw( )
{
    Xaxis2d.Draw( Black, -15, 15, THIN );
    Yaxis2d.Draw( Black, -15, 15, THIN );

    ClassCurve.Def( Blue, 100, -PI, PI );
    ClassCurve.Draw( THICK );

    Ell1.Draw( MEDIUM );
    Y = PulseY.Next( );
    Ell2.Def( Green, 100, P2d( 0, -Y ),
              P2d( B, -Y ), P2d( 0, A - Y ) );
    Ell2.Draw( MEDIUM );
}
```

We redefine the class curve c and the ellipse e_2 in each new frame and get a nice animation. As visualized in Figure 2.65, c changes its shape from an astroid (Y = 0) to Steiner's cycloid (Y = $\pm A \mp B$). In the latter case, there exists exactly one fixed point of the projectivity between e_1 and e_2, and the class of c is only three. ♢

Example 2.42. Angle stretch
Let $r = r(u)$, $\varphi = \varphi(u)$ be the parametric representation of a plane curve c in polar coordinates. We can associate the *angle-stretched curve* c^* *of factor* f with c. It is defined by the parametric representation $r^*(u) = r(u)$, $\varphi^* = f \cdot \varphi(u)$. With the help of some of Open Geometry's predefined 2D operations, we can easily visualize c^*, even if we do not have the parametric representation of c in polar coordinates.

In `"angle_stretch.cpp"` we display the angle-stretched curve of a straight line. This task is just as difficult as drawing the angle-stretched curve of an arbitrary parameterized curve. We define the straight line as a global constant and can immediately derive the parameterized equation of the angle stretched curve:

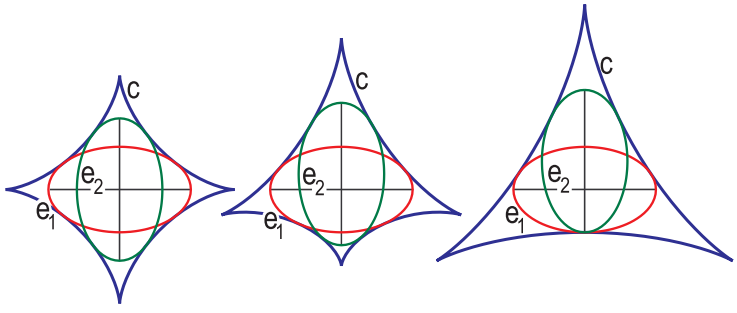

FIGURE 2.65. In general, two congruent ellipses e_1 and e_2 generate a curve of class four (left, middle). The class curve on the right-hand side (Steiner's cycloid) is of class three only, since there is one fixed point of the projective relation between e_1 and e_2.

Listing from `"angle_stretch.cpp"`:

```
const StrL2d Line( P2d( 3, 0 ), V2d( 0, 1 ) );

// the angle stretched straight line
class MyAngleStretch: public ParamCurve2d
{
public:
   void Def( Color c, int n, Real u1, Real u2, Real factor )
   {
      Factor = factor;
      ParamCurve2d::Def( c, n, u1, u2 );
   }
   P2d CurvePoint( Real u )
   {
      V2d v = Line.InBetweenPoint( tan( u ) );
      Real alpha = v.PolarAngleInDeg( );
      alpha *= Factor;
      v.Rotate( alpha );
      return P2d( 0 + v.x, 0 + v.y );
   }
private:
   Real Factor;
};
MyAngleStretch AngleStretch;
```

We want to write an animated program. Therefore, the stretch factor of c^* is an input parameter of the curves's Def(...) function. We use a pulsing real Factor for that purpose and define c^* with every single new frame:

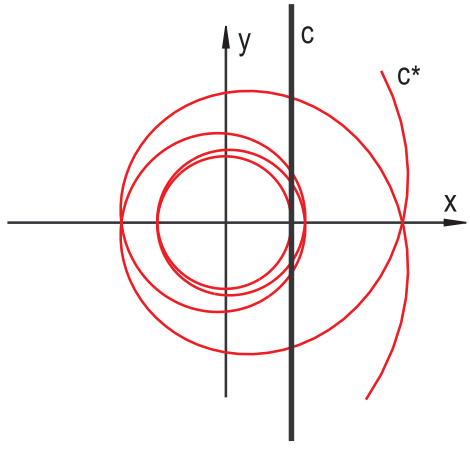

FIGURE 2.66. c^* is an angle-stretched curve to the straight line c.

Listing from `"angle_stretch.cpp"`:

```
void Scene::Draw( )
{
   // Define the angle−stretched curve. The bigger the stretch
   // factor, the more inbetween points are necessary.
   int n = ( int ) fabs( 70 * Factor( ) );
   AngleStretch.Def( Red, n + 50, −PI_2, PI_2, Factor( ) );

   // Draw everything and..
   ShowAxes2d( Black, −10, 10, −8, 8 );
   AngleStretch.Draw( THIN, 20 );
   Line.Draw( Black, −10, 10, THICK );

   //...print current stretch factor on the screen.
   PrintString( Black, 7, −5.3, "stretch factor...%2.2f",
            Factor( ) );
}
```

Note that the arc length as well as the curvature of c^* increase with the absolute value of the stretch factor. Therefore, the number n of curve points has to be chosen with respect to this value. In order to compare it with the output curve, we print the input value Factor on the screen. Finally, we do not forget to insert

```
Factor.Next( );
```

in Animate( ) before we watch the interesting spiral effect of the program on the computer screen (Figure 2.66). ♢

Refractions

Refractions are to be seen everywhere in daily life. Diving in the sea or watching fish in an aquarium produces remarkable optical effects. Refractions play an important role in technical applications as well: eyeglasses, optical lenses, underwater photography, etc. Mathematicians have been interested in refraction phenomena for quite a while, and today their geometric properties are well known:

We consider a refracting plane ϱ. It may be imagined as the surface of a calm pool of water. Light propagates with speed c_1 through the air (on one side of ϱ) and with speed c_2 through the water (other side of ϱ). The real number $r := c_1/c_2$ is called the *refraction ratio* or *index of refraction*. A straight line a_1 (incidence angle α_1 to the normal n of ϱ) is refracted into a straight line a_2 through $a_1 \cap \varrho$ with incidence angle α_2 according to *Snell's law* (Figure 2.67);

$$\frac{\sin\alpha_1}{\sin\alpha_2} = r. \tag{12}$$

That is, $\alpha_1 > \alpha_2$ iff $r > 1$. Though in principle we have $\alpha_1, \alpha_2 \in [-\pi/2, \pi/2]$, there is a restriction on either α_1 or α_2: For $r > 1$, the refracted ray will have a maximum angle of $|\alpha_2^{\max}| \le \arcsin r^{-1}$; for $r < 1$ rays are refracted only when $|\alpha_1| \le \arcsin r$. For example, for $r \approx 0.75$ (water→air) we have $|\alpha_1| \le \alpha_1^{\max} = 48.5°$. Refraction is always accompanied by (partial or total) reflection: The smaller the angle α_1 is, the less reflection occurs. For $|\alpha_1| \ge \alpha_1^{\max}$, we have total reflection on s.

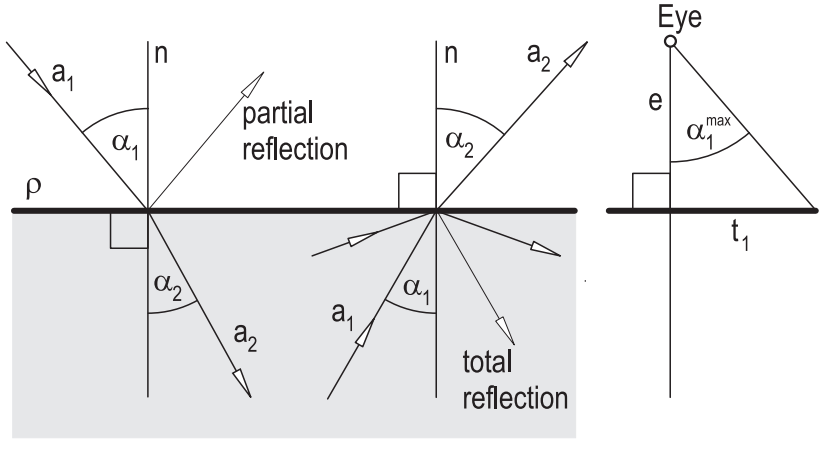

FIGURE 2.67. Snell's law (the law of refraction).

Both reflection and refraction are geometric phenomena in two dimensions only. In fact, reflection is a special case of refraction ($r = 1$). There is, however, a

small difference: While the incoming and reflected rays lie on the same side of ϱ, this is not the case with the incoming and refracted rays. Given α_1 and r, equation (12) has two theoretic solutions α_2^0 and $\pi/2 - \alpha_2^0$. Only one of them is relevant for practical purposes, and a good implementation of a refraction function in a computer program has to consider this. In the following listing you can see the implementation of the Refract(...) method of *StrL2d*.

Listing from "C/o.cpp":

```
int StrL2d::Refract( const StrL2d &refracting_line, Real ior,
    StrL2d &refracted_line ) const
{
    P2d A;
    Real lambda;
    Boolean parallel;
    SectionWithStraightLine( refracting_line, A, lambda, parallel );
    // no refraction if straight line has wrong
    // orientation or is parallel to refracting line.
    if ( lambda < 0 || parallel )
    {
        refracted_line.Def( point, dir );
        return 0;
    }

    // compute angle of outgoing ray
    // according to Snell's law
    V2d dir = GetDir( );
    V2d n = refracting_line.GetNormalVector( );

    Real alpha1 = dir.Angle( n, true, true );
    Real alpha2 = sin( alpha1 ) / ior;

    if ( fabs( alpha2 ) > 1 )
    {
        refracted_line.Def( A, dir );
        refracted_line.Reflect( refracting_line );
        return 1;
    }
    else
    {
        if ( GetPoint( ).Dist( refracting_line ) < 0 )
        {
            alpha2 *= -1;
            alpha2 = asin( alpha2 );
            n.Rotate( Deg( alpha2 ) );
            refracted_line.Def( A, n );
            return 2;
        }
```

```
        else
        {
            alpha2 = asin( alpha2 );
            n.Rotate( Deg( alpha2 ) );
            refracted_line.Def( A, -n );
            return 2;
        }
    }
}
```

The input parameters are the refracting line, the index of refraction ior, and a straight line where the refracted line will be stored. The return value is 2 if refraction occurs. It is 1 if the straight line is totally reflected and 0 if the incoming ray does not intersect the reflecting line at all (i.e., if it is oriented in a wrong direction). In order to find this ray, we check at first whether the intersection point A with the refracting line has a positive parameter value. If not, we assume that the ray points away from the refracting line. The straight line remains unchanged, and we return 0.

In the next step we compute the angles α_1 and α_2 of the incoming and the outgoing rays. If total reflection occurs, we *reflect* the straight line on the refracting line and return 1. Otherwise, we rotate the normal vector of the refracting line through α_2 and define the refracted ray. Here we must pay attention to the orientation of the refracting line in order to produce the only solution of practical relevance. Ultimately, the solution will already have the correct orientation.

Example 2.43. Refraction on a straight line
The *StrL2d* method Refract(...) can be used in many ways. For the next program (`"refraction_on_line.cpp"`) we set ourselves the following goals:

- Refract the members of a pencil of lines on a refracting line r.
- Draw the envelope d of the refracted rays (this curve is called the *diacaustic* of the pencil of lines).
- Visualize the fact that the diacaustic is the evolute of a conic section (compare, e.g., [16])

Everything will be implemented in a way that a varying index of refraction yields an animated image. With the help of a few global constants and variables...

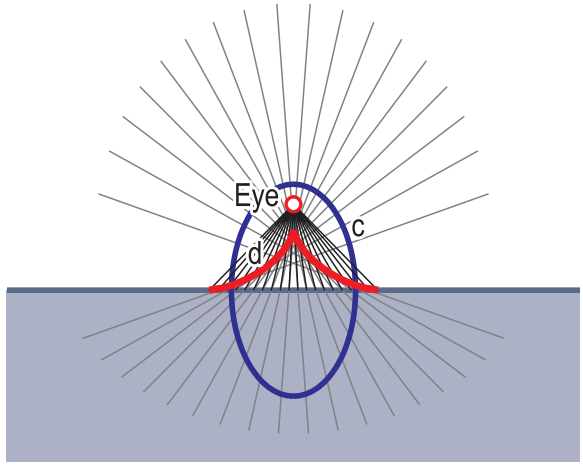

FIGURE 2.68. The diacaustic d of a pencil of lines and the characteristic conic c.

Listing from `"refraction_on_line.cpp"`:

```
const P2d Eye( 0, -2 );
const StrL2d RefrLine( P2d( Eye.x, Eye.y - 3 ), V2d( 1, 0 ) );
const Real E = Eye.Dist( RefrLine ), Tmax = 5;

PulsingReal IOR;
Conic CharConic;
Rect2d Water;
```

. . . and the OPEN GEOMETRY class *ClassCurve2d*, it is easy to implement the diacaustic d:

Listing from `"refraction_on_line.cpp"`:

```
class MyDiacaustic: public ClassCurve2d
{
public:
   StrL2d Tangent( Real t )
   {
      StrL2d s( Eye, V2d( cos( t ), sin( t ) ) );
      StrL2d refr;
      if ( s.Refract( RefrLine, IOR( ), s ) )
         return s;
      else
         return RefrLine; // dummy
```

```
    }
};
MyDiacaustic Diacaustic;
```

Just for safety reasons, we return a dummy if no real refraction occurs. In fact, we will avoid this case by choosing an appropriate parameter interval in Draw( ):

Listing from `"refraction_on_line.cpp"`:

```
// If IOR < 1 we draw only the lines
// that are not reflected totally.
Real t1, alpha_max;
if ( IOR( ) < 1 )
{
    alpha_max = asin( IOR( ) );
    t1 = E * tan( alpha_max );
}
else
{
    alpha_max = PI_2;
    t1 = Tmax;
}
Real t0 = -t1;
DrawLineCongruence( t0, t1, Black, DarkGray, 20, IOR( ) );
// compute and draw diacaustic
Diacaustic.Def( Red, 50, -alpha_max-PI_2, alpha_max-PI_2 );
Diacaustic.Draw( THICK, 100 );
```

We compute the right parameter limit t1 from formula (12) and the drawing at the right-hand side of Figure 2.67. If t1 is not restricted by physical laws, we set it to a globally declared maximal value. Then, we call a function to draw the lines of the pencil and their refraction images, define the diacaustic, and draw it. DrawLineCongruence(...) is listed below:

Listing from `"refraction_on_line.cpp"`:

```
void DrawLineCongruence( Real t0, Real t1, Color col1, Color col2, int n,
    Real ior )
{
    const Real delta = ( t1 - t0 ) / ( n -1 );
    P2d P;
    StrL2d r;
    Real t;
    int i;
    for ( i = 0, t = t0; i < n; i++, t += delta )
```

```
    {
        P = RefrLine.InBetweenPoint( t );
        StraightLine2d( col1, Eye, P, THIN );
        r.Def( Eye, P );
        int n = r.Refract( RefrLine, ior, r );
        if ( n )
            r.Draw( col2, -20, 5, THIN );
        else
        {
            P = r * RefrLine;
            r.Draw( col2, r.GetParameter( P ), 10, THIN );
        }
    }
}
```

We announced earlier that we want to visualize the fact that the diacaustic is the evolute (actually one half of the evolute) of a *characteristic conic c*. This means that the refracted rays are all normals of the characteristic conic. All we have to know is that the eye point E is a focal point and that its semiaxis is of length $a = er$, where e is the distance from eye point to refracting line and r is the current index of refraction. For $r < 1$ the characteristic conic is an ellipse, for $r > 1$ it is a hyperbola, and for $r = 1$ (reflection) it is not defined. We can implement this rather comfortably in Draw( ):

Listing from `"refraction_on_line.cpp"`:

```
if ( fabs( IOR( ) - 1 ) > 0.25 )
{
    P2d F = Eye;
    F.Reflect( RefrLine );
    P2d P;
    P = 0.5 * E * ( 1 + IOR( ) ) * Eye +
        0.5 * E * ( 1 - IOR( ) ) * F;
    CharConic.Def( Blue, 100, P, Eye, F,
                   ( IOR( ) > 1 ? HYPERBOLA : ELLIPSE ) );
    CharConic.Draw( MEDIUM );
}
```

We define the characteristic conic by two focal points Eye and F and one point P on the main axis. The construction is a little unstable in the neighborhood of $r = 1$. Therefore, we do not draw the conic within a (rather large) epsilon interval.

Finally, we implement some extra features: We change the value of the pulsing real IOR in Animate( ), print its current value on the screen, and draw a blue rectangle to symbolize the water (Figure 2.68). ♢

Example 2.44. Rainbow
Now to another program. We want to explain the genesis of a rainbow. After a shower of rain, the air is filled with small drops of water that reflect and refract the sun rays. In a certain configuration a viewer on Earth has the impression of seeing a perfect circular arc in seven colors ranging from magenta to red. The highest point of this arc is always seen under an angle of about 42°.

In order to explain the physics behind this phenomenon, we may consider the small drops of water in the air as spheres (compare Figure 2.69). A ray of light r entering a sphere is refracted (ray r_1), partially reflected on the inside (ray r_2) and refracted again as it leaves the sphere. Note that different indices of refraction apply to light of different wavelengths. Therefore, the incoming ray splits up in multiple rays whenever it is refracted.[31]

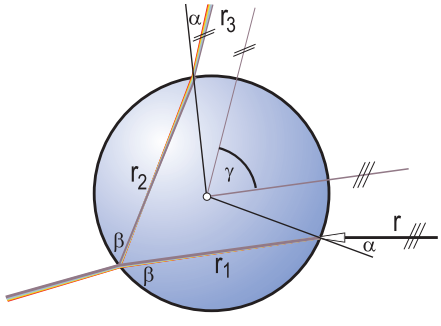

FIGURE 2.69. A ray of light r_1 is refracted and reflected in a drop of water.

In order to understand the genesis of a rainbow, we must consider the angle γ between incoming ray r and outgoing ray r_3. It turns out that there exists a maximum value of γ if r changes its position but not its direction. This is the case for sun rays; they are almost parallel.

Figure 2.70 illustrates that the maximal angle is near 42°, which confirms our observations. In this direction the intensity of light is maximal, and we see a rainbow. Inside the rainbow the sky seems to be brighter because of all the other reflected and refracted rays.

In `"rainbow.cpp"` you will find an animation that is perhaps even more lucid than Figure 2.70. There we use the following function for the refraction on a circle:

Listing from `"rainbow.cpp"`:

```
int RefractOnCircle( const StrL2d &r, const Circ2d &circ,
```

[31] This is the reason for the colors of the rainbow.

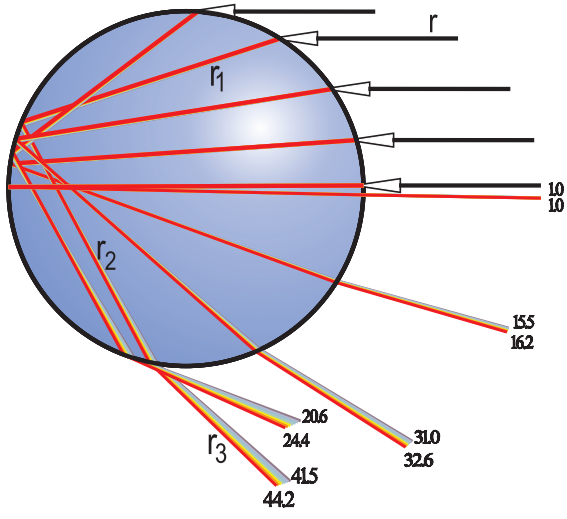

FIGURE 2.70. The maximal angle of the light rays that leave the drop of water is about 42°. In this direction the light intensity is maximal, and we can see a rainbow. The different colors are due to different indices of refraction of light rays of different wavelengths.

```
                    Real ior, StrL2d &refr_line )
{
   P2d S1, S2;
   if( !circ.SectionWithStraightLine( r, S1, S2 ) )
   {
      refr_line = r;
      return 0;
   }
   else
   {
      if ( r.GetParameter( S1 ) > r.GetParameter( S2 ) )
      {
         // swap S1 and S2 without using additional memory
         S1 = S1 - S2;
         S2 = S1 + S2;
         S1 = S2 - S1;
      }
      StrL2d tangent;
      if ( fabs( r.GetParameter( S1 ) ) < 1e-3 )
         tangent = circ.GetTangent( S2 );
      else
         tangent = circ.GetTangent( S1 );
      return r.Refract( tangent, ior, refr_line );
   }
```

```
}
```

Its input and output parameters are similar to those of *StrL2d*::Refract(...). We refract r on circ. Here, IOR is the index of refraction, and the result is stored in refr_line. The return value tells whether no reflection, total reflection, or real refraction occurs (0, 1,that and 2, respectively).

If real intersection points of r and circ exist, we arrange them in order of ascending parameter values. Then we compute the tangent in the relevant intersection point and use *StrL2d*::Refract(...). This function takes care of the correct return value as well. Note a little detail: If the first intersection point S1 has the parameter value 0 (or almost 0), we will always refract on the tangent in the second intersection point S2. This is important when two refractions occur. The new line will be defined by its direction vector and the point of refraction.

Now to the program itself. We use one circle Circle, a point L1, three straight lines Line1, Line2, Line3 and a pulsing real Phi as global constants or variables. During the whole animation Line1 will contain the fixed point L1. The circle and the pulsing real are defined in Init( ):

Listing from `"rainbow.cpp"`:

```
void Scene::Init( )
{
  const Real rad = 5;
  const P2d center( 0, 0 );
  Circle.Def( Black, center, rad, EMPTY );
  Real phi = asin( rad / L1.Distance( center ) );
  phi *= -1.001;
  Phi.Def( -PI, phi / 800, -PI - phi, -PI + phi, HARMONIC );
  ScaleLetters( 0.7, 1 );
}
```

Phi is initialized in a way that the straight line through L1 and direction (cos phi, sin phi) will always intersect the circle (phi is the current value of Phi). A very important part is Draw( ):

Listing from `"refraction_on_circle.cpp"`:

```
void Scene::Draw( )
{
   Line1.Def( L1, V2d( cos( Phi( ) ), sin( Phi( ) ) ) );
   StrL2d normal;
   Real delta = 0.0018;
   int j;
   Real ior = 0.75 - 1.2 * delta;
   Color col[7] = { DarkMagenta, LightBlue, LightCyan, LightGreen,
                    Yellow, Orange, Red };
      // seven colors of the rainbow

   for ( j = 0; j < 7; j++ )
   {
      if ( RefractOnCircle( Line1, Circle, 1/ior, Line2 ) )
      {
         P2d P = Line1 * Line2;
         if ( j == 0 )
         {
            normal.Def( P, V2d( P - Circle.GetCenter( ) ) );
            P2d H = Line1.InBetweenPoint( Line1.GetParameter( P )-5 );
            DrawArrow2d( Black, H, P, 1, 0.2, FILLED, MEDIUM );
               // mark incoming ray
         }
         if ( RefractOnCircle( Line2, Circle, ior, Line3 ) )
         {
            P = Line2 * Line3;
            P2d Q = Line2.GetPoint( );
            StraightLine2d( col[j], Q, P, MEDIUM );
            normal.Def( P, P2d(Circle.GetCenter( )) );
            P2d R = Q;
            R.Reflect( normal );
            StrL2d r( P, V2d( R - P ) );
            P2d S1, S2;
            Circle.SectionWithStraightLine( r, S1, S2 );
            if ( S1.Distance( R ) > S2.Distance( R ) )
            {
               // different way of swapping points
               P2d tmp;
               SwapElements( S1, S2, tmp );
            }
            StraightLine2d( col[j], R, S2, THIN );
            StrL2d s;
            if ( RefractOnCircle( r, Circle, ior, s ) )
               s.Draw( col[j], 0, 5, THIN );
            if ( j == 0 || j == 6 ) // print angle of outgoing ray
                                    // on the screen
```

```
                {
                    P2d H = s.InBetweenPoint( 5 );
                    char str[16];
                    sprintf( str, "%4.1f", -s.GetDir().PolarAngleInDeg( ) );
                    WriteNice( Black, str, H.x, H.y - 0.06*(j-1),
                               0, 0.6, 1, XYplane );
                }
            }
        }
        else
            Line1.Draw( Black, 0, 1000, MEDIUM );
        ior += delta;
    }
    Circle.Draw( MEDIUM );
}
```

We distinguish seven different cases according to the wavelength of the light (different color and different index of refraction). The incoming ray of light is successively refracted (or reflected) on the circle, and the respective line segments are drawn. Furthermore, we display the current angles of the outgoing rays on the screen. ♢

3

3D Graphics I

In Open Geometry, the main difference between many classes and methods in planar and spatial geometry is the suffix "3d" instead of "2d." Of course, you should know a little more than this about Open Geometry's 3D classes before you start writing programs of your own. On the one hand, geometry in three dimensions provides many more objects of interest. On the other hand, they are much harder to deal with.

As far as programming is concerned, an important difference between 2D and 3D is the velocity of animations: Numerous visibility decisions by means of z-buffering cost some CPU time. Therefore, you should use your computer's resources with care. Occasionally, we will point out examples of this.

Because of the mass of examples, we have split the chapter on 3D graphics into two parts. This part deals with the more fundamental concepts such as:

- Open Geometry's basic 3D classes,
- manipulation of the camera,
- parameterized surfaces,
- free-form surfaces (Bézier surfaces and NURBS),
- simple 3D animations.

The second part is dedicated to more complex topics.

3.1 Basic 3D Classes

In this introductory chapter we will present a few very fundamental 3D classes of OPEN GEOMETRY: points (**class** *P3d*), straight lines (**class** *StrL3d*), planes (**class** *Plane*), circles (**class** *Circ3d*), and polygons (**class** *Poly3d*). With the help of simple sample files you will see them in action and learn how to use them.

Example 3.1. Three planes
We consider a triangle and two circles in 3-space. They mutually intersect and define a unique intersection point and three line segments that we would like to display (Figure 3.1). In order to determine the intersection point and line segments, we will use the triangle's and circles' supporting planes.

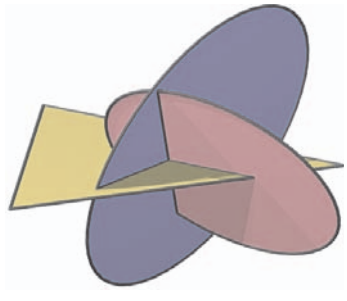

FIGURE 3.1. The mutual intersection of two circles and a triangle (compare file `"three_planes.cpp"`).

The source code of this example can be found in `"three_planes.cpp"`. We use three global variables and one global constant:

Listing from `"three_planes.cpp"`:

```
RegPoly3d R[3];      // Three regular polygons,
Plane Eps[3];        // their supporting planes,
P3d S;               // and their intersection point.

// array of colors for the three polygons
const Color Col[3] = { Yellow, Blue, Pink };
```

The triangle and the two circles will be objects of type *RegPoly3d*. We could as well use the class *Circ3d*. It would not make any difference, since it is a successor of *RegPoly3d*, and we do not need its advanced features. Furthermore, we use three planes (the polygons' supporting planes) and a 3D point S (their intersection point) as well. All elements are defined in Init( ):

Listing from `"three_planes.cpp"`:

```
void Scene::Init( )
{
   // Define the three polygons...
   int n[3] = { 3, 100, 100 };
   for ( int i = 0; i < 3; i++ )
   {
      R[i].Def( Col[i], P3d(16 - 3 * i ,0,0), Zaxis, n[i] );
      R[i].Translate( -3, 3, 1 );
   }
   // ...and bring them in a good position:
   R[0].Translate( 4, -6, 0 );
   R[1].Rotate( Zaxis, 30 );
   R[1].Rotate( Xaxis, 70 );
   R[2].Rotate( Zaxis, -50 );
   R[2].Rotate( Yaxis, -55 );

   // Define the supporting planes of the polygons...
   for ( i = 0; i < 3; i++ )
      Eps[i].Def( R[i][1], R[i][2], R[i][3] );

   // and intersect them:
   if ( !Eps[2].SectionWithTwoOtherPlanes( Eps[0], Eps[1], S ) )
      Write( "no common point" );
}
```

At first, we define the three polygons by their color, some vertex point, their axis (OPEN GEOMETRY's global constant *Zaxis*), and the number of vertex points. The two circles are approximated by polygons of 100 sides. Subsequently, we bring them into a good relative position by certain translations and rotations about the coordinate axes.

The supporting planes are defined via three polygon vertices. We can access the vertex point by means of the [] operator. The intersection point is computed by a new method of the class *Plane*: SectionWithTwoOtherPlanes(...) returns **true** if a unique intersection point exists and stores it in the global variable S. If S is not uniquely determined, we give an alert by opening a little window with the warning "no common point."

In Draw( ) we shade the three polygons in their respective colors and with a thick black contour:

Listing from `"three_planes.cpp"`:

```
void Scene::Draw( )
{
   int i, j;

   // Shade the polygons...
   for ( i = 0; i < 3; i++ )
   {
      R[i].ShadeWithContour( Black, THICK );
   }

   // ...determine the end points of their mutual intersecting
   // line segments:
   P3d S1, S2;
   for ( i = 0; i < 2; i++ )
   {
      for ( j = i + 1;  j < 3; j++ )
      {
         if ( R[i].SectionWithOtherPoly3d( R[j], S1, S2 ) )
            StraightLine3d( Black, S1, S2, THICK, 1e-2 );
      }
   }

   // Mark their intersection point. Zbuffer( false )
   // guarantees that it is always visible.
   Zbuffer( false );
   S.Mark( Black, 0.3, 0.15 );
   Zbuffer( true );
}
```

Then we have to determine the mutual intersection line segments of the polygons. For that purpose the SectionWithOtherPoly3d(...) method of *RegPoly3d* is very convenient. It returns true if an intersection occurs and stores the end points of the line segment in S1 and S2. The Open Geometry command *StraightLine3d* connects them by a thick black line.

The last argument of *StraightLine3d* is not mandatory. It gives a critical offset distance for Open Geometry's visibility algorithm (z-buffering). In our example it is larger than the default offset of 1e−3. As a consequence, the straight lines will clearly stand out. A similar trick is used for the marking of the intersection point S. It will never be concealed by any other object.

We recommend that you experiment a bit with these visibility options: Change the offset in *StraightLine3d* or mark S with active z-buffer and watch the different outputs. Note that the final output depends on your graphics card, too. For the

production of Figure 3.1 we used OPEN GEOMETRY's export option to POV-Ray. It should look a little different from your screen output, but that is all right.

More about OPEN GEOMETRY's camera option can be found in Section 3.2. The export to POV-Ray is described in Section 4.2 on page 335. ♢

Example 3.2. Some objects
OPEN GEOMETRY knows a number of geometric primitives that can be displayed with very little effort. The simple sample file `"some_objects.cpp"` is a simple sample file that demonstrates how to do this. We display a cube, a sphere, a cylinder of revolution, a tetrahedron, and a torus in aesthetic arrangement.

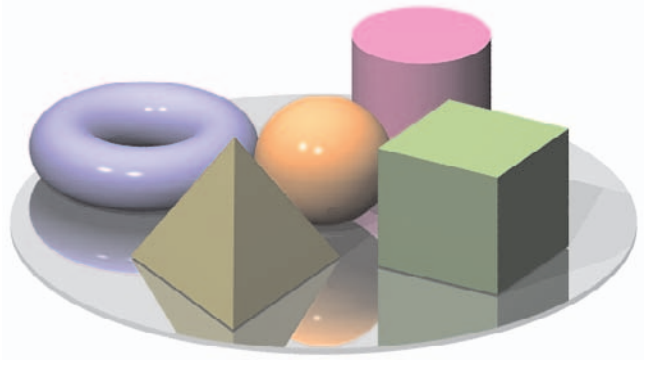

FIGURE 3.2. Geometric primitives created in OPEN GEOMETRY 2.0 and rendered with POV-Ray.

Listing from `"some_objects.cpp"`:

```
#include "opengeom.h"
#include "torus.h"    // needed for the class "Torus"

Torus T;
Sphere S;
Box B;
RegPrism Base;
RegPrism C;

class RegTetrahedron: public RegFrustum
{
public:
   void Def( Color c, Real side, const StrL3d &axis = Zaxis )
   {
      RegFrustum::Def( c, side/sqrt(3), 0, side*sqrt(2/3),
                       3, SOLID, axis );
```

```
    }
}
RegTetrahedron H;

void Scene::Init( )
{
    Base.Def( Gray, 9, 0.2, 100 );
        // regular prism of radius = 9, height = 0.2;
        // it has 100 sides, i.e., it approximates a cylindrical tray
    Base.Translate( 0, 0, −0.2 );

    C.Def( Magenta, 2, 4, 40 );
        // approximate a cylinder by a prism of 40 sides
    C.Translate( 0, −5.5, 0 );

    B.Def( Green, 4, 4, 4 ); // cube of side length 4
    B.Translate( −7, −2, 0 );

    S.Def( Orange, Origin, 2 ); // sphere of radius = 2, centered at origin
    S.Translate( 0, 0, 2 );

    T.Def( Blue, 20, 40, 2, 1 );
        // torus with 20 parallel circles and 40 meridian circles (radius = 1 );
        // the radius of the midcircle equals 2.
    T.Translate( 6, 0, 1 );

    H.Def( Yellow, 6 ); // regular tetrahedron of side length = 6
    H.Translate( 0, 5, 0 );
}
void Scene::Draw( )
{
    Base.Shade( );
    C.Shade( );
    T.Shade( );
    S.Shade( );
    B.Shade( );
    H.Shade( );
}
```

At the top of the file we declare a few global variables that describe instances of OPEN GEOMETRY primitives. An OPEN GEOMETRY class RegTetrahedron does not yet exists. Therefore, we derive it from the already existing class *RegFrustum*. A tetrahedron can be interpreted as a regular frustum of three sides where the top polygon is of radius 0. Furthermore, the side length side of the base polygon and the height height have to satisfy the relation side : height $= 1/\sqrt{3} : \sqrt{2/3}$.

Perhaps you feel that the class RegTetrahedron is useful enough to belong to OPEN GEOMETRY's standard classes. Almost certainly it will in a future version. For the time being you may implement it yourself. Section 7.4 tells you how to proceed. However, if you are a beginner, you should not worry about this more advanced topic.

In the Init( ) part we define the primitives and translate them to a good position. Note that methods for geometric transformations (Translate(...)) are available for all of the primitives we have used, even for RegTetrahedron. They inherit it from their common ancestor class *O3d*.

Figure 3.2 shows a POV-Ray rendering of the scene. We used two different light sources and assigned reflecting textures to some of the objects. ♢

3.2 Manipulation of the Camera

In this section we show how to work with the member functions of the "virtual camera." In a first example we demonstrate how the camera can be initialized and later on changed easily in the function Projection::Def( ), which has to be implemented by the user anyway. Figure 3.3 shows the output. The code is easy to comprehend. When the program is started and the animation is run, a compass is displayed from different views. When the program is restarted, a kardan joint is loaded from a file and displayed in various ortho projections.

Listing of `"manipulate_camera1.cpp"`:

```
#include "opengeom.h"

Cad3Data Obj;
Box B;
Boolean UseOrthoProj = true;

void Scene::Init( )
{
  UseOrthoProj = !UseOrthoProj;
  Obj.Delete( );
  Obj.Def( 10 );
  if ( UseOrthoProj )
  {
```

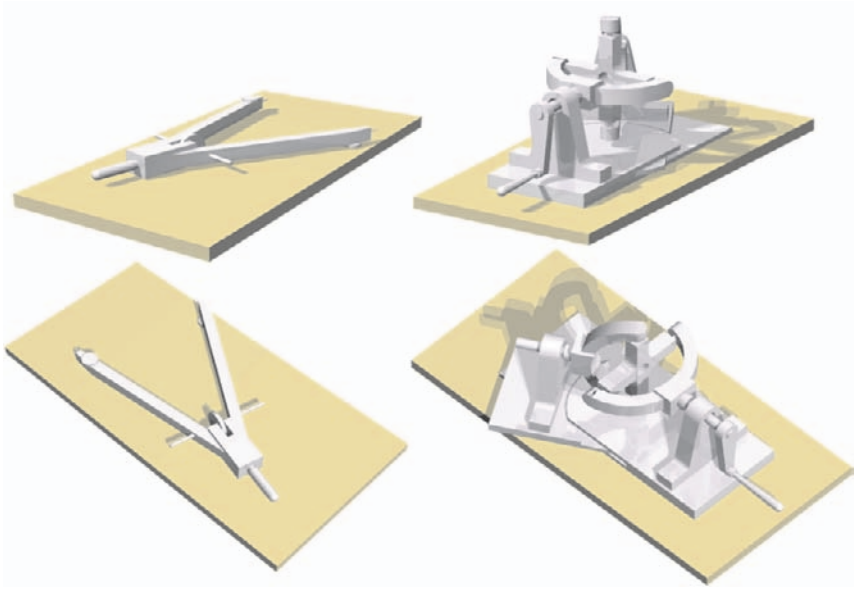

FIGURE 3.3. Different views of a scene (left: perspective, right: normal projection). Output of `"manipulate_camera1.cpp"`

```
        Obj.ReadNewMember( "DATA/LLZ/kardan_complete.llz",
                                                    0.08 );
        Obj.Translate( -5 * Xdir );
    }
    else
    {
        Obj.ReadNewMember( "DATA/LLZ/compass.llz", 0.08 );
        Obj.Rotate( Zaxis, 75 );
        Obj.Translate( 6, -2, 0.5 );
    }
    B.Def( Yellow, P3d( -8, -4, -0.5), P3d( 8, 4, 0 ) );
    AllowRestart( );
}
void Scene::Draw( )
{
    B.Shade( );
    Obj.Shade( );
}
void Scene::Animate( )
{
}
void Scene::CleanUp( )
{
}
```

```
void Projection::Def( )
{
   if ( VeryFirstTime( ) )
   {
      if ( UseOrthoProj )
         DefaultOrthoProj( 18, 18, 12 );
      else
         DefaultCamera( 18, 18, 12 );
      ZoomIn( 1.3 );
      ParallelLight( 1, 2, 3 );
   }
   else
   {
      int f = ( FrameNum( ) − 1 ) % 120;
      if ( f < 60 )
         RotateHorizontally( 2 );
      else if ( f < 70 )
         RotateVertically( 2 );
      else if ( f < 80 )
         ZoomIn( 1.02 );
      else if ( f < 90 )
         ZoomOut( 1.02 );
      else if ( f < 100 )
         RotateVertically( −2 );
      else
         LightDirection.Rotate( Zaxis, 5 );
   }
}
```

If you are a bit advanced, you already know that animations should be made only in the animation part. Thus, the manipulation of the camera should be moved to Animate( ).[1] There, however, you have to call the camera-functions — as anywhere else in your code — with the help of the global variable TheCamera.[2]

[1] If you leave the code in Projection::Def( ), the following might happen: You move to the 10-th frame and then resize the window or make some other irrelevant change that will not increase the frame number. Then the change of the camera that belongs to frame 10 will be called more than once, and this may lead to slightly different results after the animation.

[2] Actually, TheCamera is a macro that replaces the word by (*Global.camera); the global variable Global.camera is obviously a pointer variable, but don't worry about it.

Listing from "manipulate_camera2.cpp":

```
void Scene::Animate( )
{
   int f = ( FrameNum( ) - 1 ) % 120;
   if ( f < 60 )
      TheCamera.RotateHorizontally( 2 );
   else if ( f < 70 )
      TheCamera.RotateVertically( 2 );
   else if ( f < 80 )
      TheCamera.ZoomIn( 1.02 );
   else if ( f < 90 )
      TheCamera.ZoomOut( 1.02 );
   else if ( f < 100 )
      TheCamera.RotateVertically( -2 );
   else
      LightDirection.Rotate( Zaxis, 5 );
}
```

Object transformations, matrix operations

This is a good opportunity to talk about the difference between rotations of the camera and rotation of the object. When there is one single object in the scene and you want to create a certain view of it, it does not matter whether you rotate the object or the camera. The following two programs would produce very similar output. In the first program, the object (a dodecahedron) is rotated by means of a matrix so that it appears in a top view. Note that Open Geometry really changes the coordinates of the object! If you want to undo the rotation, you have to rotate the object by means of the inverse matrix!

Listing of "manipulate_camera3.cpp":

```
#include "opengeom.h"
#include "dodecahedron.h"

RegDodecahedron D;
void Scene::Init( )
{
   D.Def( Green, 4 );
   RotMatrix r;
   r.Def( TheCamera.GetPosition( ) );
   D.MatrixMult( r );
}
```

```
void Scene::Draw( )
{
   D.Shade( );
}
void Scene::Animate( )
{
}
void Scene::CleanUp( )
{
}
void Projection::Def( )
{
   if ( VeryFirstTime( ) )
   {
      DefaultOrthoProj( 28, 18, 12 );
   }
}
```

The second program is much easier to read: It creates a top view of the object without rotating it. Clearly, for this task the second solution is the better one:

Listing of "manipulate_camera4.cpp":

```
#include "opengeom.h"
#include "dodecahedron.h"

RegDodecahedron D;
void Scene::Init( )
{
   D.Def( Green, 4 );
}
void Scene::Draw( )
{
   D.Shade( );
}
void Scene::Animate( )
{
}
void Scene::CleanUp( )
{
}
void Projection::Def( )
{
   if ( VeryFirstTime( ) )
   {
      DefaultOrthoProj( 0, 0, 35 ); // Top view
   }
}
```

However, it happens quite often that you have to apply more complex rotations to a single object. This is necessary, for example, in order to let the predefined teddy touch the floor in three distinct points (Figure 3.4). The solution was found by playing around with several angles.

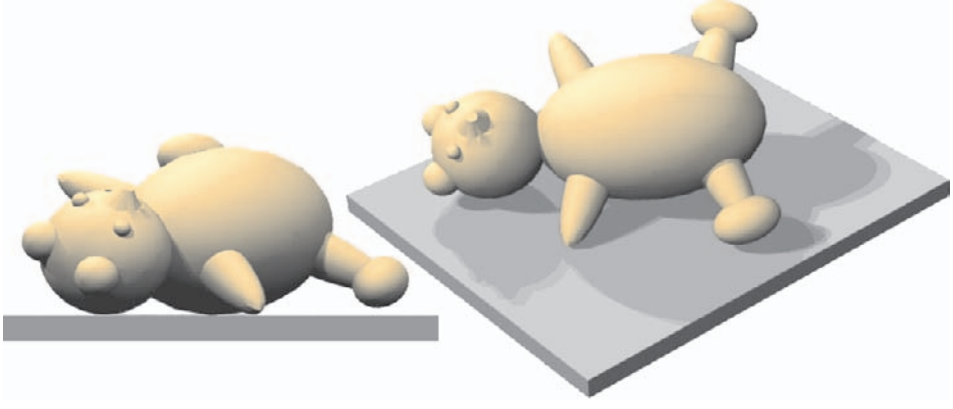

FIGURE 3.4. The teddy has to be rotated and translated to get it in the correct position with respect to the floor (output of `"manipulate_camera5.cpp"`). Left: front view, right: perspective. Note that the shadows would immediately show a wrong position.

Listing of `"manipulate_camera5.cpp"`:

```
#include "opengeom.h"

Cad3Data Teddy;
Box B;

void Scene::Init( )
{
   Teddy.Delete( );
   Teddy.Def( 10 );
   Teddy.ReadNewMember( "DATA/LLZ/teddy.llz", 0.08 );
   Teddy.Rotate( Zaxis, 60 );
   Teddy.Rotate( Xaxis, 97 );
   Teddy.Rotate( Zaxis, 50 );
   Teddy.Translate( -1, 0, 1.61 );
   B.Def( Gray, P3d( -6, -4, -0.5), P3d( 4, 4, 0 ) );
   AllowRestart( );
}
void Scene::Draw( )
{
   B.Shade( );
   Teddy.Shade( );
```

```
}
void Scene::Animate( )
{
}
void Scene::CleanUp( )
{
}
void Projection::Def( )
{
   if ( VeryFirstTime( ) )
   {
      DefaultCamera( 18, 18, 12 );
      ZoomIn( 1.3 );
      ParallelLight( 1, 2, 3 );
   }
}
```

Remember that the coordinates are manipulated with every rotation. So when you have an object with 10 000 vertices and you apply three rotations, it is definitely faster to create one single rotation matrix:

```
...
Teddy.ReadNewMember( "DATA/LLZ/teddy.llz", 0.08 );
RotMatrix z1, x, z2;
z1.Def( Arc( 60 ), Zdir );
x.Def( Arc( 97 ), Xdir );
z2.Def( Arc( 50 ), Zdir );
Teddy.MatrixMult( z1 * x * z2 );
...
```

Note that the definition of a matrix requires the angle in arc length, which is a bit different from our usual conventions, but these definitions are used quite often internally, and since the computer calculates in arc lengths, this speeds up the code a bit.

Speeding up time-critical code with OpenGL display lists

Speaking of speed and speaking of rotation matrices we can assert that both are strength of OpenGL. So, for some applications, you can use these OpenGL features in order to get a real-time animation done.

Example 3.3. Rocking horse
As an example, we want a rocking horse (Figure 3.5) to "rock realistically." The object is stored in a file, and it has a few thousand vertices. It has to be rotated about a general axis and, additionally, to be translated for each new frame. If you do this with the OPEN GEOMETRY routines, this costs a great deal of CPU time. Additionally, the changes have to be undone after the drawing. In such a case, a classical OPENGL solution is to be preferred: We use the matrix calculus with its functions *glMatrixMode*( ), *glPushMatrix*( ), and *glPopMatrix*( ). The general rotation is done with a combination of *glTranslated*( ) and *glRotated*( ), and the ordinary translation again with *glTranslated*( ). The changes are applied only to the OPENGL projection and not to the object coordinates. Thus, they are undone when we reactivate the previous matrix. Additionally, the speed is increased when we use display lists that are embedded in the *glNewList*( )–*glEndList*( ) block. With *glCallList*( ) they are redisplayed with respect to the actual matrix. Here is the complete listing of the corresponding program:

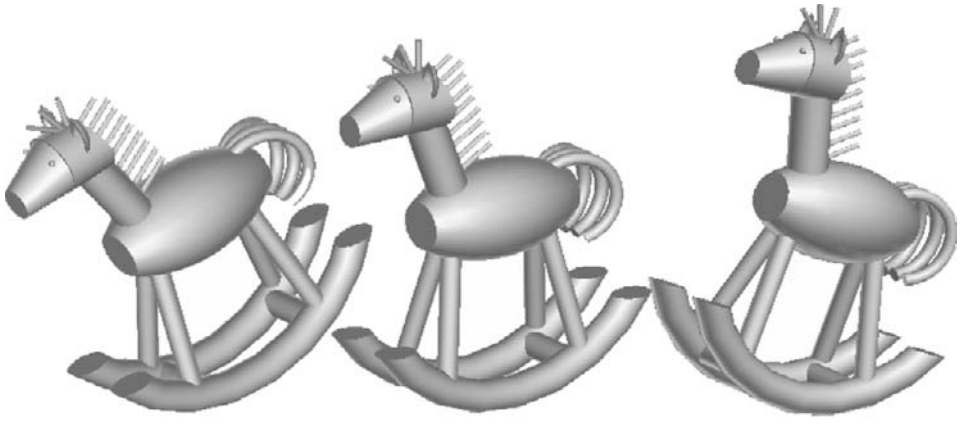

FIGURE 3.5. Rocking horse, animated in real time via OPENGL. Output of `"rocking_horse.cpp"`

Listing of `"rocking_horse.cpp"`:

```
#include "opengeom.h"

Cad3Data Horse;
PulsingReal Angle;
Boolean FirstTime;

void Scene::Init( )
{
   FirstTime = true;
   Horse.ReadNewMember( "data/llz/rocking_horse.llz" );
```

```
    Horse.SetAllColors( Brown );
    Angle.Def( 0, 2, 25, −25, HARMONIC );
    AllowRestart( );
}
void Scene::Draw( )
{
    Real t = Angle( );
    glMatrixMode( GL_MODELVIEW );
    glPushMatrix( );
    glTranslated( 0, 0, −1 );
    glRotated( t, 0, 1, 0 );
    glTranslated( 1.0 ∗ Arc( t ), 0, 0.0006 ∗ t ∗ t );
    if ( FirstTime )
    {
        glNewList( 1, GL_COMPILE_AND_EXECUTE );
        Horse.Shade( ALL_FACES );
        glEndList( );
        FirstTime = false;
    }
    else
        glCallList( 1 );
    glPopMatrix( );
}
void Scene::CleanUp( )
{
}
void Scene::Animate( )
{
    Angle.Next( );
}
void Projection::Def( )
{
    if ( VeryFirstTime( ) )
    {
        DefaultOrthoProj( 7, 9, 5 );
        SetDepthRange( −4, 4 );
        ParallelLight( 3, 5, 3 );
    }
}
```

The global variable Angle of type *PulsingReal* is a perfect tool for the creation of a harmonic movement. Note the other global variable FirstTime, which enables us to build and redisplay the list. ♢

Optimize OpenGL's hidden surface algorithm

In Projection::Def( ) of `"rocking_horse.cpp"` we used the camera's member function SetDepthRange(...), which has influence on the image quality. Its parameters should — in the optimal case — be the minimal and maximal distance from the projection plane. (This plane is perpendicular to the principal ray and contains the target point.) Everything closer to the eye point or further away is clipped by the OpenGL "graphics engine".

OpenGL uses the method of depth-buffering (or z-buffering) for the fast removal of hidden surfaces. In pure OpenGL code you would enable and disable the buffer with the following lines:

```
glEnable( GL_DEPTH_TEST );
glDepthFunc( GL_LEQUAL );
glDepthRange( 0, 1 );
...
glDisable( GL_DEPTH_TEST );
```

In Open Geometry, this is done by the scene's member function Zbuffer( ) with a *Boolean* parameter on_off. You do not need to know the algorithm in detail, but you should know that the closer its ranges are adapted to the depth range of the actual scene the better it works. And this is what SetDepthRange(...) does for you. When your scene has a diameter of 0.5 units and you set the range from −1000 to +1000, the output will be poor. Typical Open Geometry scenes have a diameter of 5 to 20 units, – like the objects on the sketch on your drawing paper. Therefore, the default depth range is from −40 to +40, which turns out to be a good average depth range.

All Open Geometry routines are optimized for the default depth range. For example, lines on a surface are by default drawn with a tiny offset in the direction to the eye point. This offset can generally be changed, when you add an additional *Real* offset number at the end of Open Geometry functions like *StaightLine3d*(...), *StrL3d*::Draw(...), *StrL3d*::LineDotted(...), *L3d*::Draw(...), *ParamSurface*::WireFrame(...), *Polyhedron*::Contour(...), etc. The default value for offset is the value *STD_OFFSET* = 0.001. For extreme cases, you can choose a value offset = 10 * *STD_OFFSET* or so for large scenes, or offset = 0.1 * *STD_OFFSET* for tiny scenes. You can let Open Geometry do the work for you when you use the scene's member function ChangeDefaultOffset( ). For example, a parameter value 2 doubles the value of *STD_OFFSET*. In the rare case that you deal with surface parts that have almost zero distance, the algorithm will cause problems in any case.

For the production of Figure 3.6 we used Open Geometry's default depth range and offset. The two surfaces (Plücker conoids; compare Example 3.17) are apparently identical. However, for the picture on the left-hand side we zoomed

in on a very small surface (diameter of 1 to 2 drawing units). You can see the bad visibility performance in the regions around the surface axis. The object on the right-hand side has a diameter of about 20 drawing units. There, the hidden line algorithm works well.

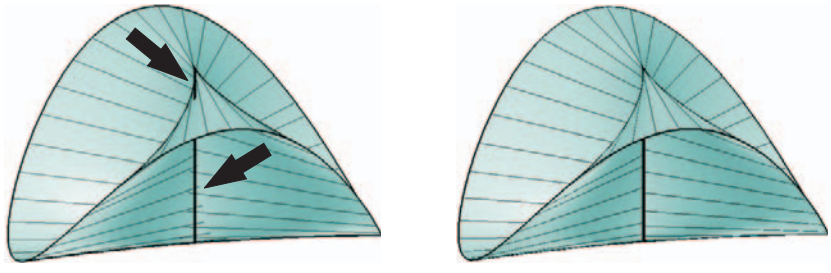

FIGURE 3.6. Too small surfaces without additional settings of default offset and depth range may cause visibility problems. The surface on the left-hand side has a diameter of about 1 to 2 drawing units; the surface on the right-hand side has good OPEN GEOMETRY dimensions

How to find the optimal view and lighting quickly

As a final point, we want to explain how you can quickly optimize the view and the light direction for your application: You first make a guess (say, take the default camera from `"defaults3d.h"`). Then you start the program and — via keyboard shortcuts or menu — change the camera and the light source. When you are satisfied with the output, save the settings via menu or the shortcut <`Ctrl + l`>. This creates a file `"tmp.dat"` that looks like this:

```
// The following changes have been made interactively
// Insert this code into 'Projection::Def( )'
    if ( VeryFirstTime( ) )
    {
        DefaultCamera( 42.76, 13.35, 21.40 );
        ChangeTarget( 7.93, -1.20, 3.90 );
        ChangeFocus( 70.0 );
        SwitchToNormalProjection( );
        SetDepthRange( -10.4, 10.4 );
        ParallelLight( 0.27, -0.53, 0.80 );
    }
```

Just insert the contents of this file into your application file in your implementation of Projection::Def( ). The file `"tmp.dat"` is written to OPEN GEOMETRY's

standard output directory as specified in `"og.ini"`. If you want to change this value, you can edit this file and insert a path of your choice.[3]

Impossibles

This chapter's concern is how to display 3D objects on a 2D medium like the screen of your computer or a sheet of paper coming out of your printer, a very common task indeed. However, it can be a bit tricky. For the transition from space to plane, we always pay with a loss of information: Theoretically, it is impossible to reconstruct a 3D object from a single image in 2D.

When we watch real-life objects, this problem usually does not occur. We have two images as input data (we watch with two eyes), and we get additional information by walking around the object, turning our head or rolling our eye-balls. Furthermore, we consider phenomena of illumination and reflection, and last but not least, we have a certain notion of the object we watch.

Reversed, all of this can be used to create illusions on your computer screen. It is you who determines what the user will see. You can play with the user's imagination and make her/him believe to be seeing something that is not really there. This is, however, not just a joke. Deceiving images may be used to create sensible impressions. We will present a few (not too serious) examples of optical illusions in the following.

Example 3.4. Tribar
Start the program `"tribar.cpp"`. You will see three lengthy boxes connected in a way to form three right angles. Of course, this is not possible, and you can immediately discover the trick by (literally) taking a different point of view (Figure 3.7).

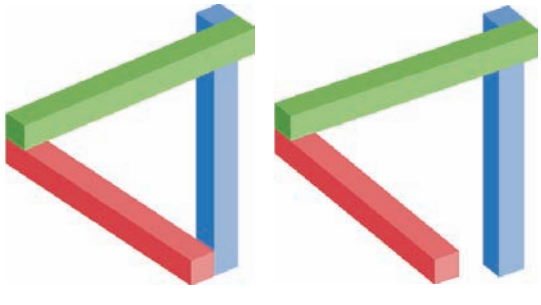

FIGURE 3.7. The impossible tribar.

[3]This may turn out to be necessary in case of a network installation of OPEN GEOMETRY.

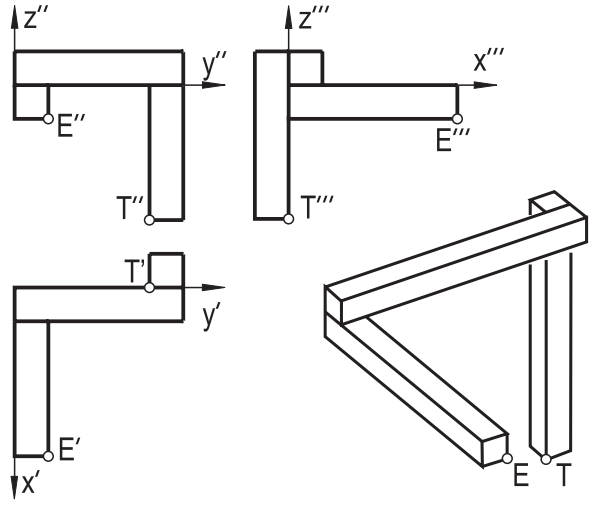

FIGURE 3.8. The arrangement of the boxes and the position of eye and target point.

The corresponding code is very easy. We arrange three boxes parallel to the coordinate axes in the way that is displayed in Figure 3.8.[4] The boxes are of a certain dimension B × B × H. Thus, the points E and T that — seemingly — coincide in Figure 3.7 have coordinates $(h, b, 0)^t$ and $(0, h-b, 2b-h)^t$, respectively. Now we have only to find the right projection.

Listing from `"tribar.cpp"`:

```
void Projection::Def( )
{
   if ( VeryFirstTime( ) )
   {
      DefaultCamera( H, B, 0.5 * H );
      ChangeTarget( 0, H - B, -H + 2 * B + 0.5 * H );
      ChangeFocus( 30 );
      SwitchToNormalProjection( );
      ParallelLight( 0.72, 0.35, 0.60 );
   }
}
```

[4]You can probably imagine the scene after a short look at the right-hand side of Figure 3.7. This is, however, possible only because you automatically assume boxes with edges parallel to the coordinate axes. Otherwise, the reconstruction problem has no unique solution.

We choose E and T as eye point and target point, change the focus, and switch to normal projection.[5] This yields projection rays parallel to ET and ensures that these points have coinciding images. It is important to use normal projection. Otherwise, the different distortions of the boxes would immediately give the clue to the solution. ♢

Example 3.5. Reconstruction
Another example of a misleading 2D image of a 3D object can be found in `"reconstruction.cpp"`. In the first frame you will see an object that you would probably interpret as the image of a wire frame model of a cube. The corresponding 3D object has, however, eight random vertices on certain projection rays. Rotate the camera a little and you will see its irregular shape (Figure 3.9).

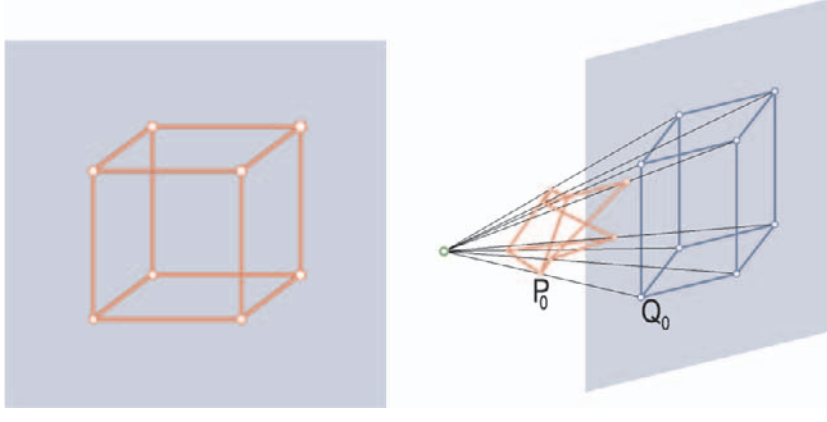

FIGURE 3.9. The image of a bizarre object seems to be the image of a cube.

Again, the code is not difficult. We define a global constant Eye of type *P3d*. Eye is a point on the positive x-axis. Our image plane will be $[y, z]$. After initializing the vertices P[i] of the "cube's" image we create the "cube's" vertices in Init( ):

Listing from `"reconstruction.cpp"`:

```
const Real alpha = 1;
for ( i = 0; i < 8; i++ )
{
    V3d dir = Eye - P[i];
    dir.Normalize( );
    Q[i] = P[i] + rnd( ) * dir;
    Rad[i] = Eye.Distance( Q[i] ) * tan( Arc( alpha ) );
```

[5] In fact, before doing this we translated E and T by $h/2$ in the z-direction in order to keep the image at the center of the screen.

```
}
```

We choose random numbers that yield points Q[i] between the image plane and the eye point. At the same time, we compute the radius Rad[i]. Later in the Draw( ) part we will mark the point Q[i] with radius Rad[i] and thus ensure that these points seemingly have the same size in the first frame (their corresponding fovy angle is $2 \cdot \text{alpha}^\circ$). This is important for the illusion of a normal projection. In reality we use, of course, the central projection with center Eye. ♢

Example 3.6. Impossible cube
A third example of an optical illusion is implemented in `"impossible_cube.cpp"` (Figure 3.10). The corresponding source code is too low-level to be of interest at this place. We do nothing but include a CAD3D-object: the impossible cube. The difficult task is the creation of this cube.

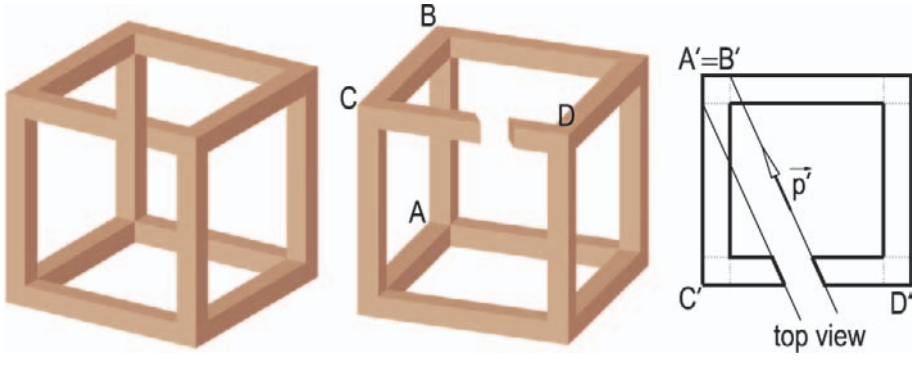

FIGURE 3.10. The impossible cube and its construction.

Again, we have to use a normal projection. After deciding on the direction $\vec{p}$ of the projection rays, we have to define the "projection cylinder" of the vertical edge at the back left. By subtracting this cylinder from the horizontal edge at the front top, we cut out the gap to look through. This needs only some elementary vector calculus. Any computation can be performed in 2D only. ♢

Example 3.7. Mirrors
Having a quick look at Figure 3.11, you probably have the impression of seeing twelve balls in cyclic arrangement. Perhaps a second glance will convince you that there are only two balls. The remaining ten arise from multiple reflection on two mirror planes. Starting the program `"deception.cpp"` will confirm this. You can rotate the eye point in the horizontal and vertical directions; the impression will remain the same. Still, everything is just illusion.

Reflections in real-time animations are usually a difficult task. As long as only planar reflections occur, the introduction of a "mirror window" is a good idea.

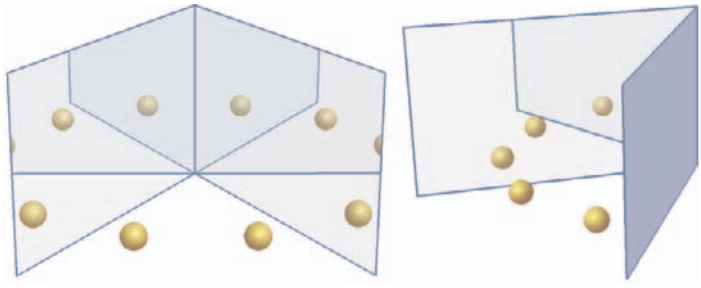

FIGURE 3.11. How many balls do you see?

Instead of computing the reflection image in the mirror, one can watch a virtual scene through a window (Figure 3.12).

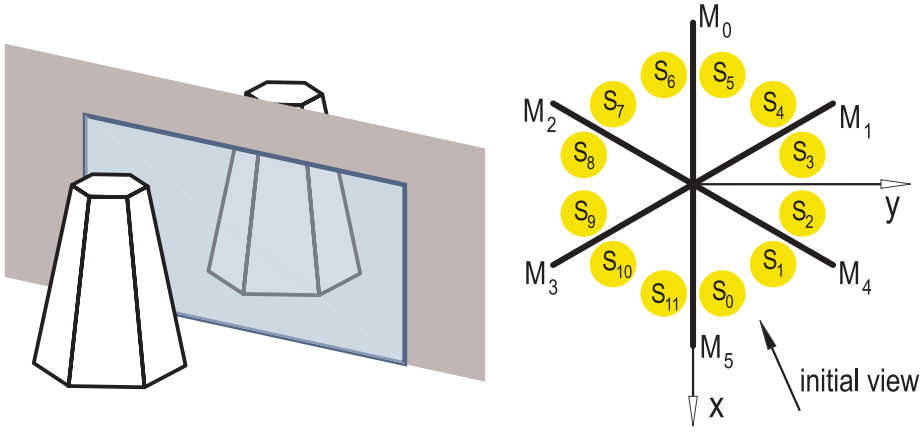

FIGURE 3.12. Looking through a mirror window (left). The arrangement of mirrors and spheres (right).

In `"deception.cpp"` we used this trick in a slightly modified way. Actually, we display no fewer than *twelve* balls and *six* mirror frames. In order to give the illusion of mirrored images, we draw additional polygons that prevent the spectator from seeing too much. In contrast to Figure 3.12, these polygons are shaded in pure white.

In order to reach a satisfying result, we must take into account many little things. We are to begin with the drawing order of the mirrors. They are defined in Init( ):

Listing from `"deception.cpp"`:

```
// define the six mirrors
```

```
const Real l = 12, w = 7;
int i;
for ( i = 0; i < 6; i++ )
{
    Mirror[i].Def( Blue, l, w, FILLED );
    Mirror[i].Rotate( Xaxis, 90 );
}
// it is important to define
// the mirrors in this order.
Mirror[0].Rotate( Zaxis,  180 );
Mirror[1].Rotate( Zaxis,  120 );
Mirror[2].Rotate( Zaxis, -120 );
Mirror[3].Rotate( Zaxis, - 60 );
Mirror[4].Rotate( Zaxis,   60 );
```

Later, in Draw( ), we will shade them using transparency. Therefore, it is important that the frontmost mirrors be the last to be shaded. We already took care of this when defining them. In Draw( ) we can shade them in ascending order if we look at the scene from an eye point with only positive x and y coordinates. The twelve spheres are initialized in Init( ) as well:

Listing from `"deception.cpp"`:

```
const Real r = 0.5, dist = 0.7 * l;
for ( i = 0; i < 12; i++ )
{
   S[i].Def( Yellow, Origin, r, 50 );
   S[i].Translate( dist, 0, r );
   S[i].Rotate( Zaxis, 15 + i * 30 );
}
```

In order to give you some orientation for the following, we display the mirrors and spheres together with their numbers in a top view on the right side of Figure 3.12. Initially (in the first frame), we will look at the scene in the direction indicated in the picture; i.e., both mirror surfaces M_4 and M_5 are visible. In this case, we have to draw two large polygons (*deceptors*) with two "windows" to cover the parts of the scene that must not be visible. In Figure 3.13 we display the outline of the deceptor polygons without shading them.[6]

The deceptors are objects of type *ComplexPoly3d* (because they are not convex). We define them in Init( ):

[6] Actually, we had to decrease their size considerably in comparison with the original program. Otherwise, they would not be visible.

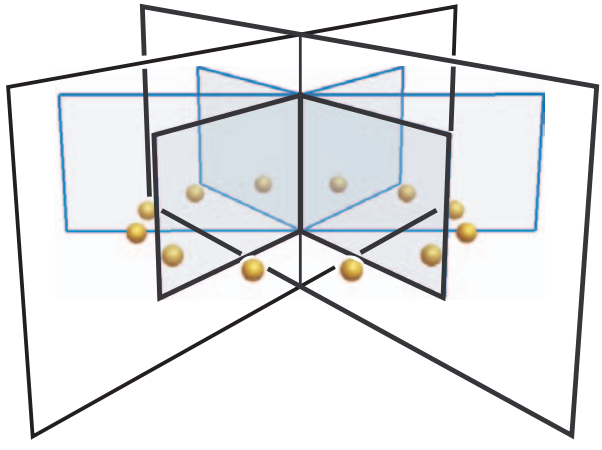

FIGURE 3.13. The deceptor polygons.

Listing from `"deception.cpp"`:

```
// define the two deceptive polygons
const Real m = 3000, v = 1500; // cant be too large
Poly3d poly[2];
poly[0].Def( NoColor, 4 );
poly[0][1].Def( m, 0, v );
poly[0][2].Def( -m, 0, v );
poly[0][3].Def( -m, 0, w - v );
poly[0][4].Def( m, 0, w - v );
poly[1].Def( NoColor, 4 );
poly[1][1].Def( l, 0, 0 );
poly[1][2].Def( l, 0, w );
poly[1][3].Def( 0, 0, w );
poly[1][4].Def( 0, 0, 0 );
Deceptor[0].Def( PureWhite, 2, poly );
Deceptor[1].Def( PureWhite, 2, poly );
Deceptor[1].Rotate( Zaxis, 60 );
```

Now the most important part has to be done. With respect to the current position of the eye point, we have to draw different parts of the scene. In the initial configuration we must shade all mirrors transparently, shade all spheres, and draw both deceptors. If the eye point is rotated between mirrors four and one (compare Figure 3.12), we must shade the mirrors three and five with transparency, while mirror one is displayed in a different color and without transparency (we just see its back). Likewise, we must shade only the spheres $S_8, \ldots, S_{11}$, S_0, S_1 and deceptor zero. Using two variables Draw4 and Draw5 of type *Boolean*, we can distinguish all four cases of relevance. The complete listing of Draw( ) reads as follows:

Listing from `"deception.cpp"`:

```
void Scene::Draw( )
{
   // get the position of the eye with
   // respect to Mirror[4] and Mirror[5]
   Eye = TheCamera.GetPosition( );
   if ( Eye.DistFromPlane( Pi4 ) >= 0 )
      Draw4 = true;
   else
      Draw4 = false;
   if ( Eye.DistFromPlane( Pi5 ) <= 0 )
      Draw5 = true;
   else
      Draw5 = false;

   int i;

   if ( Draw4 && Draw5 ) // Mirror[4] and Mirror[5] visible
   {
      for ( i = 0; i < 12; i++ )
         S[i].Shade( );
      for ( i = 0; i < 6; i++ )
         Mirror[i].ShadeTransparent( LightBlue, 0.2,
            DarkBlue, MEDIUM );
      Deceptor[0].Shade( );
      Deceptor[1].Shade( );
   }
   else if ( Draw4 ) // Mirror[4] visible, Mirror[5] invisible
   {
      for ( i = 0; i < 6; i++ )
         S[i].Shade( );
      Mirror[1].ShadeTransparent( LightBlue, 0.2,
         DarkBlue, MEDIUM );
      Mirror[4].ShadeTransparent( LightBlue, 0.2,
         DarkBlue, MEDIUM );
```

```
        Mirror[5].ShadeWithContour( DarkBlue, MEDIUM );
        Deceptor[1].Shade( );
    }
    else if ( Draw5 ) // Mirror[5] visible, Mirror[4] invisible
    {
        S[0].Shade( );
        S[1].Shade( );
        for ( i = 8; i < 12; i++ )
            S[i].Shade( );
        Mirror[3].ShadeTransparent( LightBlue, 0.2,
            DarkBlue, MEDIUM );
        Mirror[5].ShadeTransparent( LightBlue, 0.2,
            DarkBlue, MEDIUM );
        Mirror[4].ShadeWithContour( DarkBlue, MEDIUM );
        Deceptor[0].Shade( );
    }
    else // Mirror[4] and Mirror[5] invisible
    {
        S[0].Shade( );
        S[1].Shade( );
        Mirror[4].ShadeWithContour( DarkBlue, MEDIUM );
        Mirror[5].ShadeWithContour( DarkBlue, MEDIUM );
    }
}
```

♢

3.3 A Host of Mathematical Surfaces

Nowadays, scientific publications or courses at school or university require excellent illustrations and pictures. One of the most important tasks in this context is the visualization of mathematical surfaces. Open Geometry has some very convenient methods to give you support with this:

- You can easily draw parameterized curves and shade parameterized surfaces.
- There exist features that allow you to draw the wire frame, border lines, or contours of surfaces.
- You have access to many objects of geometric relevance like the points, normal vectors, or the tangent planes of your surface. Furthermore, you can easily write your own methods to create additional objects satisfying the needs of your specific surface. In this section you will find many examples for this.

- You can intersect two different surfaces or calculate the self-intersection of a parameterized surface.
- You have the possibility of creating smoothly shaded surfaces with soft highlights. A new feature is the *surface with thickness* that yields images of high perspicuity.
- You can rotate the surface in real time and watch it from any viewpoint. Thus, you will get a clear notion even of complicated surfaces.
- You can produce high-quality printouts of your surface.
- You can export the surface to the freeware ray-tracing program POV-Ray. There, you can equip it with different textures and assign different material properties to it.

Example 3.8. Rider's surface
A first example demonstrates the power of OPEN GEOMETRY's class *ParamSurface*. We will visualize a special ruled surface Φ that was called *Reiterfläche* (*rider's surface*) in [8]: Let d_1 and d_2 be two straight lines with orthogonal direction vectors that do not intersect. In an appropriate coordinate system their equations may be written as

$$d_1 \ldots y = 0,\ z = B \qquad \text{and} \qquad d_2 \ldots x = 0,\ z = -B.$$

Now, the generators of Φ are the straight lines that intersect d_1 and d_2 in points D_1 and D_2 of a fixed distance $D > 2B$. Using the abbreviation $A := \sqrt{D^2 - 4B^2}$, a parametric representation of Φ reads

$$\Phi\colon X(u,v) \ldots \vec{x}(u,v) = (1-v)\vec{d_1}(u) + v\vec{d_2}(u),$$

where $\vec{d_1}(u) = (A\cos u, 0, B)^t$ and $\vec{d_2}(u) = (0, A\sin u, -B)^t$. The parameter range is, of course, $[-\pi, \pi] \times (-\infty, \infty)$. Now the implementation of Φ in OPEN GEOMETRY is easy (`"normal_surfaces.cpp"`).

Listing from `"normal_surfaces.cpp"`:

```
class RuledSurf: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real u, Real v )
   {
      return ( 1 - v ) * D1( u ) + v * D2( u );
   }
};
RuledSurf RuledSurface;
```

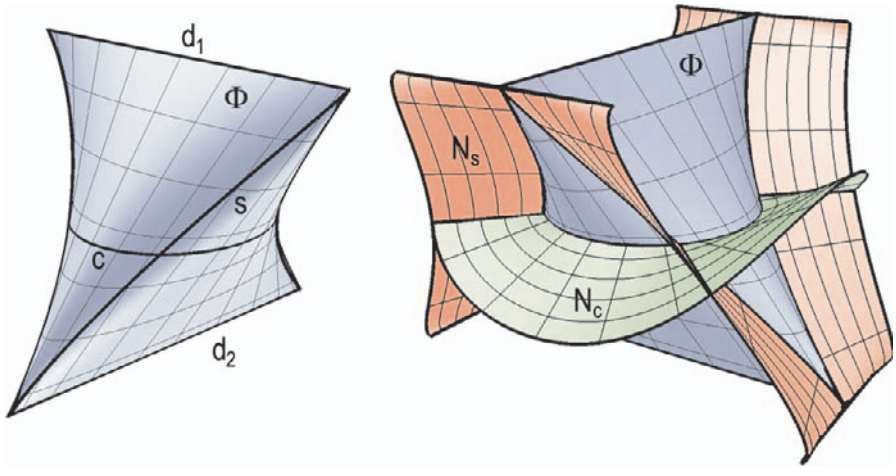

FIGURE 3.14. Φ, the base circle c and the line of striction s (left); Φ and the normal surfaces along c and s (right).

So far, everything is standard as presented in [14]. But perhaps you want to do something special with your surface (see Figure 3.14). For example, you can display the normal surface N_c of Φ along the *base circle* $c \ldots v = 1/2$.[7] Without any computation we implement it as follows:

Listing from `"normal_surfaces.cpp"`:

```
class NormSurfBaseCircle: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real u, Real v )
   {
      return RuledSurface.SurfacePoint( u, 0.5 ) +
           v * RuledSurface.NormalVector( u, 0.5 );
   }
};
NormSurfBaseCircle NormalSurface_BC;
```

[7]That is, the ruled surface generated by the normals of Φ along c.

In the geometric theory of ruled surfaces, the *line of striction* $s\colon S(u)\ldots\vec{s}(u)$ plays an important role. We determine the feet $S_0(u,\varepsilon)$ and $S_1(u,\varepsilon)$ of the common normal of two neighboring generators $g(u)$ and $g(u+\varepsilon)$. Passage to the limit $\varepsilon\to 0$ yields the limiting *point of striction*

$$S(u)=\lim_{\varepsilon\to 0}S_0(u,\varepsilon)=\lim_{\varepsilon\to 0}S_1(u,\varepsilon)$$

on $g(u)$. The direct computation of the line of striction is usually a difficult task. With Open Geometry it can be visualized almost immediately:

Listing from `"normal_surfaces.cpp"`:

```
class LineOfStric: public ParamCurve3d
{
public:
   P3d CurvePoint( Real u )
   {
      const Real eps = 1e-5;
      const Real u0 = u - eps, u1 = u + eps;

      StrL3d g0, g1;
      g0.Def( D1( u0 ), D2( u0 ) );
      g1.Def( D1( u1 ), D2( u1 ) );

      P3d G0, G1;
      V3d dummy;
      g0.CommonNormal( g1, dummy, G0, G1 );

      return 0.5 * ( G0 + G1 );
   }
};
LineOfStric LineOfStriction;
```

We determine two "symmetric" generators $g(u\pm\varepsilon)$ of Φ, use the Open Geometry method `CommonNormal(...)` of *StrL3d* to get the feet S_0 and S_1 of the common normal, and return their midpoint (Figure 3.14).[8]

Finally, we want to display the normal surface N_s of Φ along the line of striction s. This is a little harder, since we do not have the parametric representation $S(u)=X(u,v(u))$ of s. That is, we cannot use the `NormalVector(...)` method of *ParamSurface*. Instead, we write a function to compute the normal vector directly. The rest is the same as above.

[8]There exists an alternative to this procedure that is even simpler. We could define Φ as an instance of the class *RuledSurface*. There, all these methods are directly available (compare pages 211 ff).

Listing from "normal_surface.cpp":

```
V3d NormalVector_LoS( Real u )
{
    V3d normal_vector =
    V3d( D1( u ), D2( u ) ) ^ LineOfStriction.TangentVector( u );
    normal_vector.Normalize( );
    return normal_vector;
}

class NormSurfLoS: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real u, Real v )
   {
      return LineOfStriction.CurvePoint( u ) +
           v * NormalVector_LoS( u );
   }
};
NormSurfLoS NormalSurface_LoS;
```

Just a few more remarks:

- The v-lines of Φ and the normal surfaces are straight lines. Thus, we do not need too many facets in the v-direction to get a satisfying picture. We used a total 100×20 facets; if you do not need the contour outline, even 100×10 is enough. This will considerably speed up any animation.
- We encounter numerical problems if we use the parameter range $u \in [-\pi, \pi]$ for the normal surface along the line of striction s. The "normal" returned by NormalVector_LoS(u) is not defined at the cusps of s. For this reason, we defined the surface as follows:

Listing from "normal_surfaces.cpp":

```
const Real eps = 0.05;
NormalSurface_LoS.Def( LightRed, 100, 10,
                  u1 + eps, u2 - eps, 0, 4 );
```

- The normal surfaces have four straight lines in common. We compute and draw them as follows:

Listing from "normal_surfaces.cpp":

```
StrL3d s( RuledSurface.SurfacePoint( PI / 4, 0.5 ),
          RuledSurface.NormalVector( PI / 4, 0.5 ) );
int i;
for ( i = 0; i < 4; i++ )
{
   s.Rotate( Zaxis, 90 * i );
   s.Draw( Black, -4, 0, THICK );
}
```

The complete output can be seen in Figure 3.14. ♢

We already mentioned the new OPEN GEOMETRY class *RuledSurface*. The following pages will show you how to use it. In fact, Example 3.8 could have used that class as well. But sometimes it is better to learn things the hard way.

Ruled Surfaces

Ruled surfaces play an important role in theoretical as well as in applied geometry. They consist of a one-parameter family of straight lines (the *generating lines*, *generators*, or *rulings*). We can always parameterize a ruled surface Φ according to

$$\Phi \ldots \vec{X}(u,v) = \vec{x}(u) + v\vec{y}(u).$$

The curve $d \ldots \vec{x} = \vec{x}(u,v)$ is called the *directrix* of Φ; the *direction vector* $\vec{y}(u)$ gives the direction of the generator $g(u)$. The directrix is not unique; it can be replaced by an arbitrary curve on Φ (Well, you had better not take a generator!). For theoretical investigations one usually assumes that $|\vec{y}| \equiv 1$. This is, however, not necessary for our purposes.

OPEN GEOMETRY's class *RuledSurface* is derived from *ParamSurface*. It is an *abstract* class with two purely virtual member functions. In order to use *RuledSurface* you have to derive your own additional class. The purely virtual member functions of *RuledSurface* are

```
virtual P3d DirectrixPoint( Real u );
virtual V3d DirectionVector( Real u );
```

Since their meaning is clear, we can immediately start with an easy example.

Example 3.9. Right helicoid
If we choose directrix and direction vector according to

$$\vec{x}(u) = \begin{pmatrix} 0 \\ 0 \\ pu \end{pmatrix} \quad \text{and} \quad \vec{y}(u) = \begin{pmatrix} \cos u \\ \sin u \\ 0 \end{pmatrix},$$

we get a right helicoid. In "right_helicoid.cpp" we derive the class MyRuledSurface:

Listing from "right_helicoid.cpp":

```
class MyRuledSurface: public RuledSurface
{
public:
  virtual P3d DirectrixPoint( Real u )
  {
    Real p = 1;
    return P3d( 0, 0, p * u );
  }
  virtual V3d DirectionVector( Real u )
  {
    V3d v( cos( u ), sin( u ), 0 );
    return v;
  }
};
MyRuledSurface Helicoid;
```

The definition in Init( ) is identical to the definition of a general parameterized surface:

Listing from "right_helicoid.cpp":

```
void Scene::Init( )
{
  int n = 3;
  int n1 = n * 35;                  // Segments in the u-direction.
  int n2 = 15;                      // Segments in the v-direction;
  Real u1 = -n * PI, u2 = n * PI; // Range of parameter u.
  Real v1 =  -5, v2 = -v1;          // Range of parameter v.
  Helicoid.Def( LightCyan, n1, n2, u1, u2, v1, v2 );
  Helicoid.PrepareContour( );
}
```

The same is true of the shading of the helicoid. It is done in Draw( ) and reads thus:

Listing from `"right_helicoid.cpp"`:

```
void Scene::Draw( )
{
   Helicoid.Shade( SMOOTH, REFLECTING );
   Helicoid.Contour( Black, MEDIUM );
   Helicoid.WireFrame( Black, 9, 24, THIN );
   Helicoid.DrawBorderLines( Black, MEDIUM );
   Helicoid.DrawDirectrix( Black, 2, MEDIUM );
}
```

The only new method is DrawDirectrix(...). The second parameter gives the number of points on the curve. In our case, the directrix is a straight line (the z-axis), and two points are sufficient. Note that we need not worry about the v-parameter range or the pitch of the screw. DrawDirectrix(...) will automatically draw the right interval of the z-axis. ♢

Example 3.10. Ruled surface of order three
Sometimes a ruled surface is not defined by directrix and direction vector but in a geometric way. In [26] we find, e.g., the following exercise:

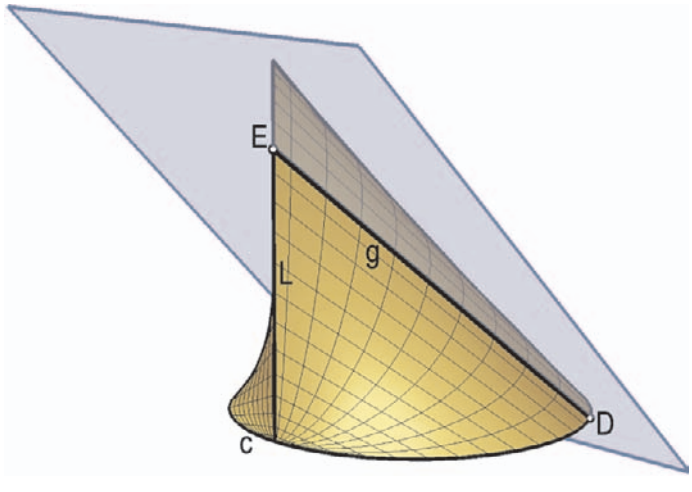

FIGURE 3.15. The generation of the ruled surface Φ.

Draw the ruled surface Φ whose generators:

- intersect a circle c in a plane parallel to $[x, y]$,

- intersect a straight line L parallel to z through a point of c,
- are parallel to the plane $\pi \ldots x = z$.

The resulting ruled surface Φ is algebraic of order three. Thus, we called the corresponding OPEN GEOMETRY program `"order_three_surface.cpp"`. We will write the code in a way that is independent of the particular positions of L and π. If L does not intersect c, the resulting surface is of order four. In this case, you should probably rename the program to avoid misunderstandings.

At the top of `"order_three_surface.cpp"` we define the radius of the circle c, the straight line L, and the normal vector V of π. Furthermore, we will use a rectangle Frame and a real U0 to illustrate the generation of the surface with an animation.

Listing from `"order_three_surface.cpp"`:

```
const Real Radius = 6;
const StrL3d L( P3d( Radius, 0, 0 ), V3d( 0, 0, 1 ) );
const V3d V( 1, 0, -1 );
Rect3d Frame;
Real U0;
```

The interesting thing in the implementation of Φ as a ruled surface is the virtual function DirectionVector(...). We do not perform any calculations but construct the direction vector directly. We define the plane through a directrix point and perpendicular to V and intersect it with L:

Listing from `"order_three_surface.cpp"`:

```
virtual V3d DirectionVector( Real u )
{
  P3d D = DirectrixPoint( u );
  Plane p( D, V );
  P3d X = p * L;
  V3d v;
  v = X - D;
  return v;
}
```

In order to illustrate the generation of Φ = Phi, we want to draw the rectangle Frame parallel to π through a generating line g(u0). In Init( ) we prepare this by positioning the rectangle parallel to π:

Listing from "order_three_surface.cpp":

```
Real distx = 2;
Real disty = 5;
Real angleVZ = Deg( V.Angle( V3d( 0, 0, 1 ) ) );
Real length = Phi.SurfacePoint( PI, 0 ).
    Distance( Phi.SurfacePoint( PI, 1 ) );
Frame.Def( Blue, length + 2 * distx,
          2 * Radius + 2 * disty, FILLED );
StrL3d axis( Origin, V3d( 0, 0, 1 ) ^ V );
Frame.Rotate( axis, angleVZ );
```

Later, in Draw( ), we will translate the rectangle to its proper position. But first, we shade Φ and draw the generating line g(U0):

Listing from "order_three_surface.cpp":

```
P3d D = Phi.DirectrixPoint( U0 );
P3d E = Phi.SurfacePoint( U0, 1 );
StraightLine3d( Black, D, E, THICK, 1e-3 );

Real c = cos( U0 );
Real s = sin( U0 );
Real u1 = ArcTan2( V.y * V.y * s + 2 * V.x * V.y * c
          - V.x * V.x * s, V.y * V.y * c - 2 * V.x * V.y * s
          - V.x * V.x * c );
P3d F = Phi.DirectrixPoint( -u1 + PI );
StraightLine3d( Black, F, E, THICK, 1e-3 );
```

In fact, we do a little more. The supporting plane of Frame intersects Φ in three straight lines: g(U0), a line at infinity, and a second generator g(U1). We draw the second generator as well. The intersection point F of g(U1) and c is determined analytically, as displayed above. It is the second intersection point of the straight line through D parallel to *V3d*(−V.y, V.x, 0) and the circle c. Now to the adjustment of the rectangle:

Listing from "order_three_surface.cpp":

```
Real t0 = 0.45, t1 = 0.5 - t0;
P3d X = t0 * ( Frame[2] + Frame[3] ) +
    t1 * ( Frame[1] + Frame[4] );
P3d Y = 0.5 * ( D + F );
V3d v = Y - X;
Frame.Translate( v );
```

We compute a point X inside the frame, the midpoint Y of D and F, and translate the rectangle by the vector $\overrightarrow{XY}$. The actual position of X varies with the shape of the surface Φ. If you change the straight line L or the vector V, you may have to use another real number t0. Finally, we vary U0 in Animate( ) to get some animation as well. ♢

Example 3.11. Elliptical motion in 3D
Another example of a ruled surface defined by geometric constraints rather than mathematical equations is displayed in "3d_ell_motion.cpp". Let d_1 and d_2 be two nonintersecting straight lines in space, and let $D_1 \in d_1$ and $D_2 \in d_2$ be two points on them. If we vary them on d_1 and d_2 without changing their distance, we get a series of straight lines D_1D_2 that define a ruled surface Φ.[9]

Because of its simple mechanical definition, Φ is of importance in 3D kinematics and robotics. It is closely related to the general elliptic motion in the plane (compare "ellipse2.cpp").

In "3d_ell_motion.cpp" we start by setting some parameters that define the two straight lines d_1 and d_2. In an appropriate coordinate system their equations are

$$d_1 \ldots \begin{cases} y = 0, \\ z = -H/2, \end{cases} \qquad \text{and} \quad d_2 \ldots \begin{cases} y = \tan(\varphi)\, x, \\ z = H/2, \end{cases}$$

respectively (compare Figure 3.16). Then we introduce a parameter R that gives the constant distance of D_1 and D_2 during the motion. Given the point $D_1 \in d_1$, there exist two corresponding points D_2 and $\overline{D}_2$ on d_2.[10] We get them as intersection points of d_2 and the sphere with center D_1 and radius R. They may, however, be conjugate complex or coinciding. The latter happens for two points D_1^l and D_1^r on d_1. They are situated at distance $d = \sqrt{R^2 - H^2}/\sin(\varphi)$ from the foot $M_1(0, 0, -H/2)^t \in d_1$ of the common normal of d_1 and d_2.

These obvious and clear considerations are a little tricky to implement: The class *RuledSurface* does not allow two direction vectors. We write a function of our own to handle this task (R = Radius, $H/2$ = H2, $\cos\varphi$ = CAngle, $\sin\varphi$ = SAngle):

Listing from "3d_ell_motion.cpp":

```
P3d CalcPoint( Real t, Boolean sign )
{
    P3d M( Dist * cos( t ), 0, -H2 );
    Sphere s( NoColor, M, Radius + 1e-8 );
    StrL3d l( P3d( 0, 0, H2 ), V3d( CAngle, SAngle, 0 ) );
```

[9] If the directions of d_1 and d_2 are orthogonal, we get the *rider's surface* of Example 3.8.

[10] Because of this, d_1 and d_2 are double lines of Φ.

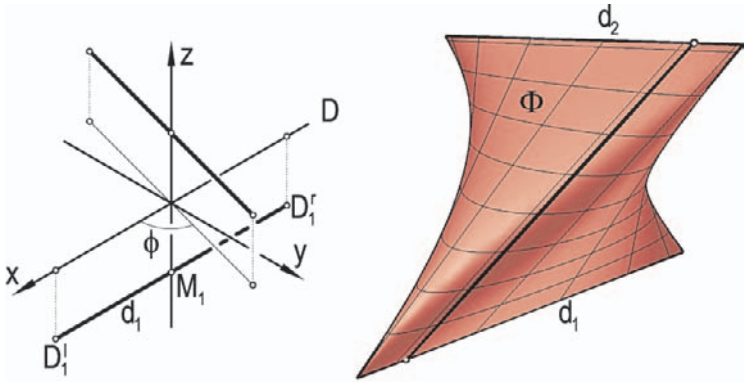

FIGURE 3.16. Relevant points and lines (left) and the ruled surface Φ (right).

```
    P3d A( 0, 0, 0 ), B( 0, 0, 0 );
    int n = s.SectionWithStraightLine( l, A, B );
    if ( n )
    {
        if ( sign )
            return ( l.GetParameter( A ) >
                l.GetParameter( B ) ? A : B );
        else
            return ( l.GetParameter( A ) >
                l.GetParameter( B ) ? B : A );
    }
    else
    {
        Write( "warning:suspicious surface points!" );
        return Origin;
    }
}
```

The first input parameter t defines a point $D_1(d\cos t, 0, -H/2)^t$ on d_1. Using the cosine function for the parameterization of d_1 allows us to avoid the points with imaginary generators while we reach the remaining points twice. CalcPoint(...) will return one of the two corresponding points on d_2 according to the second input parameter sign. The use of sign helps us to find the correct return value later in the program.

Two points D_2 and $\overline{D}_2$ corresponding to $D_1 \in d_1$ belong — as points on d_2 — to two parameter values v_2 and $\overline{v}_2$. We choose the bigger parameter value if sign is true and the smaller value if sign is false. One may imagine that sign == true always picks out the "left" or "upper" solution point.

The implementation of the ruled surface is now not too difficult:

Listing from "3d_ell_motion.cpp":

```
class MyRuledSurface: public RuledSurface
{
public:
   virtual P3d DirectrixPoint( Real u )
   {
      return M1 + Dist * cos( u ) * Dir1;
   }
   virtual V3d DirectionVector( Real u )
   {
      P3d D = DirectrixPoint( u );
      P3d X = CalcPoint( u, u < 0 );
      return X - D;
   }
};
MyRuledSurface RS;
```

If the directrix parameter u is negative, we decide on the "left" solution point. If it is positive, we take the other one. Since we use the parameter intervals $u \in [-\pi, \pi]$, $v \in [0, 1]$, this will yield the desired results. You can test it by watching the animation of "3d_ell_motion.cpp". A generator of Φ glides around the surface without changing its length. ♢

Example 3.12. A special torse
As our next example we choose a very special surface T, the tangent surface of the space curve c with parameterized equation

$$\begin{aligned} x(t) &= -\sqrt[4]{3}(\sin 3t + 3\sin t), \\ y(t) &= \sqrt[4]{3}(\cos 3t + 3\cos t), \\ z(t) &= 6\sin t. \end{aligned}$$

The surface T is a torse, a developable ruled surface and at the same time the hull surface of a one-parameter set of planes. The curve c is called the *edge of regression* of T. We can state the following interesting properties of c and T (compare [19], Figure 3.17):

- c is a sextic with two real cusps corresponding to the parameter values $t = \pm\pi/2$.
- c lies on an *ellipsoid of revolution* E with semiaxis lengths of $a = 4\sqrt[4]{3}$ and $b = 4\sqrt{3}$. E is centered around the origin and z is its axis of revolution.
- The top view of c is a *nephroid* c', i.e., a special trochoid.

- T is an algebraic surfacealgebraic!surface of order six and class four.
- The rulings of T (i.e., the tangents of c) intersect $[x, y]$ in a second nephroid n and have *constant slope* $\gamma = \arctan(3^{-\frac{1}{4}}) \approx 37.2°$ with respect to $[x, y]$.
- There exist two conic sections on T: a double hyperbola h in $[x, z]$ and an ellipse e in a plane ε with slope γ through y.

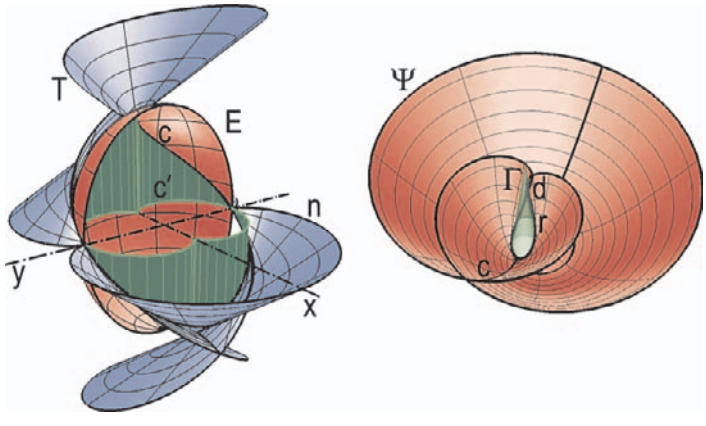

FIGURE 3.17. The torse T and related curves and surfaces.

Our aim is now the production of a high-quality picture of this surface that shows all of its relevant properties (`"torse.cpp"`). We start by deriving a successor of the class *RuledSurface*:

Listing from `"torse.cpp"`:

```
class MyTorse: public RuledSurface
{
public:
  virtual P3d DirectrixPoint( Real t )
  {
    Real x, y, z;
    x = -pow( 3, 0.25 ) * ( sin( 3 * t ) + 3 * sin( t ) );
    y =  pow( 3, 0.25 ) * ( cos( 3 * t ) + 3 * cos( t ) );
    z =  6 * sin( t );
    return P3d( x, y, z );
  }
  virtual V3d DirectionVector( Real t )
  {
    Real x, y, z;
    x = - 3 * pow( 3, 0.25 ) * ( cos( 3 * t ) + cos( t ) );
    y = - 3 * pow( 3, 0.25 ) * ( sin( 3 * t ) + sin( t ) );
```

```
        z = 6 * cos( t );
        V3d v;
        v.Def( x, y, z );
        StrL3d s;
        P3d D, S;
        D = DirectrixPoint( t );
        s.Def( D, v );
        S = XYplane * s;
        return V3d( S.x - D.x, S.y - D.y, S.z - D.z );
    }
};
MyTorse Torse;
```

Note that we do not use the tangent vector of c directly as direction vector of the rulings. In DirectionVector(...) we rather intersect the ruling with $[x, y]$ and return the vector connecting directrix point and intersection point. Hence, the nephroid n will be the borderline of the surface if we use the v-parameter interval [v0, 1]. Now we define the projection cylinder of c and the ellipsoid of revolution E:

Listing from `"torse.cpp"`:

```
class NephroidCylinder: public ParamSurface
{
public:
    virtual P3d SurfacePoint( Real u, Real v )
    {
        P3d A = Torse.DirectrixPoint( u );
        P3d B( A.x, A.y, 0 );
        return ( 1 - v ) * A + v * B;
    }
};
NephroidCylinder Cylinder;

class MyEllipsoid: public ParamSurface
{
public:
    virtual P3d SurfacePoint( Real u, Real v )
    {
        const Real a = 4 * pow( 3, 0.25 ) + 0.05, b = 4 * sqrt( 3 );
        P3d P( a * cos( u ), 0, b * sin( u ) );
        P.Rotate( Zaxis, v );
        return P;
    }
};
MyEllipsoid Ellipsoid;
```

The double hyperbola h is not entirely contained in the real surface part. In Figure 3.17 only small parts are visible. Because of this, we use an *L3d* object DoubleHyperbola.[11] We determine its points by intersecting the rulings of T with $[x, z]$. Only if the intersection points are within the parameter range of T we add them to DoubleHyperbola:

Listing from `"torse.cpp"`:

```
int n = 28;
DoubleHyperbola.Def( Black, n );
int i;
Real t, delta = PI / ( n - 1 );
for ( i = 0, t = -0.5 * PI; i < n; i++, t += delta )
{
   StrL3d s = Torse.Generator( t );
   P3d S = XZplane * s;
   if ( fabs( fabs( s.GetParameter( S ) ) ) <=
        Torse.DirectionVector( t ).Length( ) )
   DoubleHyperbola[i] = S;
}
```

A similar method is suitable for the ellipse on T. Here it is sufficient to determine five ellipse points:

Listing from `"torse.cpp"`:

```
P3d P[5];
Plane p;
Real alpha = atan( pow( 3, -0.25 ) );
p.Def( Origin, V3d( sin( alpha ), 0, cos( alpha) ) );
for ( i = 0; i < 5; i++ )
   P[i] = p * Torse.Generator( i );
TorseEllipse.Def( Black ,150, P );
```

[11] At the time we wrote `"torse.cpp"` the method GetSelfIntersection(...) of *ParamSurface* was not yet available. Therefore, this example shows a work-around to the problem of finding the self-intersection in a special case. Occasionally, this may be quite useful.

Finally, in order to draw the nephroid c' we get the borderline of the projection cylinder using the GetULine(...) method of *ParamSurface*. Taking a sensible amount of surface points and curve points, we can draw the image. You have to decide whether you want a fast animation or a really beautiful picture (e.g., an illustration for a book on OPEN GEOMETRY). In the latter case, you can draw contour outline (which is very time-consuming) and compute a relatively high number of facets on each surface.

The surface displayed in this example is related to another interesting surface Ψ, a surface with a one-parameter family of congruent parabolas on it. The parabolas are the lines of steepest slope, and their supporting planes envelop just the torse T. The surface Ψ possesses an edge of regression r that is situated on a cylinder of revolution Γ, a double cubic c, and a double hyperbola h. We display everything in Figure 3.17.

The corresponding OPEN GEOMETRYcode is standard. We use parameterized equations of the surfaces and curves. Perhaps it is noteworthy that we drew the v-lines (parabolas and, at the same time, curves of steepest slope) of the surface separately:

Listing from `"surface_of_parabolas.cpp"`:

```
L3d vline1, vline2;

Surface.GetVLine( vline1, 0.5 * PI, Black, 30 );
Surface.GetVLine( vline2, 1.5 * PI, Black, 30 );

vline1.Draw( MEDIUM, 0.0005 );
vline2.Draw( MEDIUM, 0.0005 );

int i, n = 25;
Real u, delta = 2 * PI / ( n - 1 );
for ( i = 0, u = 0; i < n; i++, u += delta )
{
    Surface.GetVLine( vline1, u, Black, 30 );
    Surface.GetVLine( vline2, u + PI, Black, 30 );
    vline1.Draw( THIN );
    vline2.Draw( THIN );
}
```

We did this because we used a parametric representation where two v-lines cover one parabola. If we used the standard WireFrame(...) or VLines method of *ParamSurface*, it would yield only half-parabolas in this case. Of course, we need to know that corresponding v-lines belong to parameter values v_0 and $v_0 + \pi$, respectively. ♢

Intersections and self-intersections of surfaces

When it comes to surfaces, one thing must not be overlooked: Surfaces intersect each other, and sometimes they intersect themselves. Few programs manage to determine intersection lines correctly, and even fewer manage to determine the self-intersections. This disadvantage is diminished by hidden-surface removal. Nevertheless, a real "perfect image" needs the lines to be drawn as well. What is more, sometimes one has to do further calculations with the intersection lines, and at least at that point we definitely need an intersection algorithm!

Open Geometry has a rather robust built-in algorithm for the determination of intersection curves. The result is of type *ComplexL3d*. Such a complex line is an accumulation of an arbitrary number of (closed or not closed, straight or curved) lines. With the Draw(...) command it is displayed easily.

Example 3.13. Intersection of parameterized surfaces
Let us learn by doing: The following program `"surf_x_surf.cpp"` displays two intersecting parameterized surfaces (tori). The output is to be seen in Figure 3.18. Transparency helps to get a better imagination.

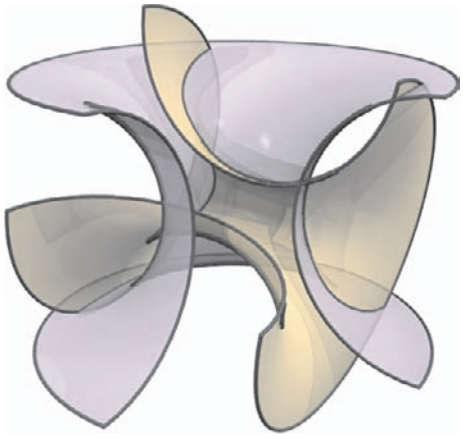

FIGURE 3.18. Two intersecting surfaces (output of `"surf_x_surf.cpp"`).

Note that you should call the member function PrepareContour( ) of the surfaces. This internally creates an "intelligent" approximating polyhedron that "knows" about neighboring triangles, etc. As an additional advantage, you get the contour line "for free." Obviously a shaded image with border lines, contour lines, and intersection lines supports human imagination considerably.

Listing of `"surf_x_surf.cpp"`:

```
#include "opengeom.h"
```

```
#include "defaults3d.h"
class Type1: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real u, Real v )
   {
      const Real a = 9, b = 6;
      Real r = a + b * cos( v );
      P3d P( r * cos( u ), r * sin( u ), b * sin ( v ) );
      return P;
   }
};
class Type2: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real u, Real v )
   {
      const Real a = 7.5, b = 6;
      Real r = a + b * cos( v );
      P3d P( r * cos( u ), r * sin( u ), b * sin ( v ) );
      P.Rotate( Xaxis, 80 );
      return P;
   }
};
Type1 Surf1;
Type2 Surf2;
ComplexL3d Section;

void Scene::Init( )
{
   int n1 = 61, n2 = 25;
   Real u1 = PI/2, u2 = 2 * PI; // Range of parameter u
   Real v1 = PI / 2, v2 = v1 + PI; // Range of parameter v
   Surf1.Def( LightOrange, n1, n2, u1, u2, v1, v2 );
   Surf1.PrepareContour( );
   Surf2.Def( LightOrange, n1, n2, u1, u2, v1, v2 );
   Surf2.PrepareContour( );
   Surf1.SectionWithSurface( Section, Surf2, false );
   ChangeDefaultOffset( 2 );
}
void Scene::Draw( )
{
   Section.Draw( Black, MEDIUM );
   Surf2.Shade( SMOOTH, REFLECTING );
   Surf1.Shade( SMOOTH, REFLECTING );
   Surf2.Contour( Black, MEDIUM );
   Surf2.DrawBorderLines( Black, MEDIUM );
   Surf1.Contour( Black, MEDIUM );
   Surf1.DrawBorderLines( Black, MEDIUM );
}
```

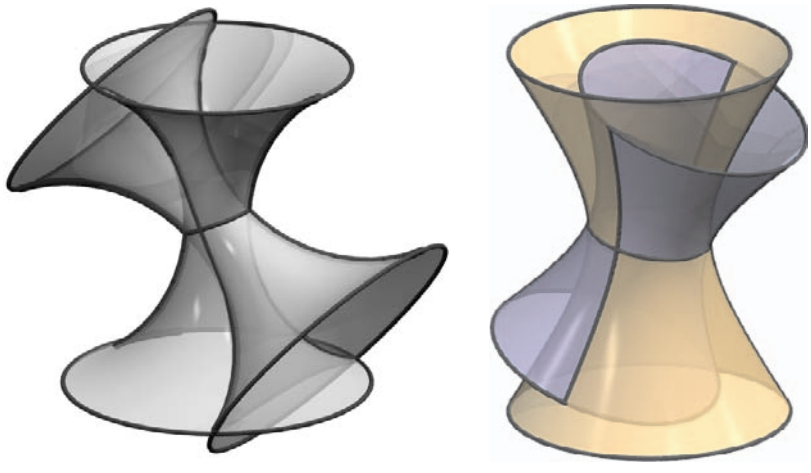

FIGURE 3.19. Two touching intersecting surfaces. Left: two tori, right: two hyperboloids (output of `"hyp_x_hyp.cpp"`).

Every mathematician knows that things get critical when the surfaces touch each other. Then the intersection line has a double point, and nearly all algorithms become unstable. Open Geometry sometimes overcomes the problem by itself (Figure 3.19; left), sometimes an easy trick helps: In Figure 3.19, right, one of the two surfaces was rotated about its axis about half a degree. The reason for this is quite simple: The surfaces are approximated by polyhedra, i.e., they are triangulated. Two almost identical triangles may induce a numerical problem. A tiny rotation helps in most cases. ♢

Example 3.14. Confocal quadrics
Figure 3.20 shows an example that is anything but trivial. The lines of curvature on a general ellipsoid are calculated as the intersection lines with "general hyperboloids." For the correct display, the program not only has to calculate the intersection lines of the hyperboloids with the ellipsoid but of course also the intersection line of the two different types of hyperboloids. At the very least, the two-sheeted hyperboloid has therefore to be split up. As a matter of fact, since everything is so symmetrical, it is enough to consider only one sheet; the intersection line of the other sheet is congruent. For more details, please have a look at the corresponding program `"lines_of_curvature.cpp"`.

Just a few passages shall be pointed out. We declare the ellipsoid, a certain number of the two types of hyperboloids, and a global parameter A that changes the curvature of the hyperboloids:

Listing from `"lines_of_curvature.cpp"`:

```
const int N = 6, N0 = 4;
Type1 Ell;
Type2 Hyp1[N];
Type3 Hyp2[N];
ComplexL3d S1[N], // Hyp1[i] ∩ Ell
   S2[N], // Hyp2[i] ∩ Ell
   S3;  // Hyp1[i] ∩ Hyp2[i]
Real A;
```

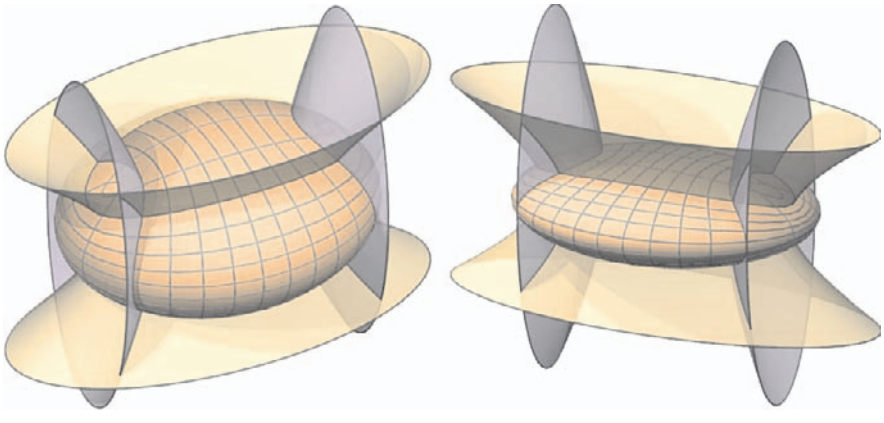

FIGURE 3.20. The lines of curvature on a general ellipsoid are space curves of order four. They are the intersecting lines of the ellipsoid with confocal quadrics, i.e., hyperboloids (output of `"lines_of_curvature.cpp"`).

In the initialization part, the surfaces and the intersection lines are stored:

Listing from `"lines_of_curvature.cpp"`:

```
Real delta = 0.71;
A = delta / 2;
Real v2 = 1.25;
for ( i = 0; i < N; i++ )
{
   Hyp1[i].Def( LightBlue, n1, n2, 0, 2*PI, 0.01, v2 );
   Hyp1[i].SectionWithSurface( S1[i], Ell, false );
   Hyp2[i].Def( LightGreen, n1, n2, 0, 2*PI, 0.01, v2 );
   Hyp2[i].SectionWithSurface( S2[i], Ell, false );
   if ( i == N0 )
```

```
    {
        Hyp1[N0].SectionWithSurface( S3, Hyp2[N0], false );
    }
    A += delta;
}
```

In the drawing part, finally, everything is displayed:

Listing from `"lines_of_curvature.cpp"`:

```
int i;
for ( i = 0; i < N; i++ )
{
    S1[i].Draw( i == N0 ? Black : Gray, MEDIUM );
    S1[i].Rotate( Zaxis, 180 );
    S1[i].Draw( i == N0 ? Black : Gray, MEDIUM );
    S1[i].Rotate( Zaxis, 180 );
    S2[i].Draw( i == N0 ? Black : Gray, MEDIUM );
    S2[i].Rotate( Yaxis, 180 );
    S2[i].Draw( i == N0 ? Black : Gray, MEDIUM );
    S2[i].Rotate( Yaxis, 180 );
}
Ell.Shade( SMOOTH, REFLECTING );
Ell.Contour( Black, MEDIUM );
for ( i = 0; i < 2; i++ )
{
    S3.Draw( Black, MEDIUM );
    S3.Rotate( Zaxis, 180 );
    S3.Draw( Black, MEDIUM );
    S3.Rotate( Yaxis, 180 );
    Hyp1[N0].Shade( SMOOTH, REFLECTING );
    Hyp1[N0].Contour( Black, MEDIUM );
    Hyp1[N0].ULines( Black, 2, MEDIUM );
    Hyp1[N0].Rotate( Zaxis, 180 );
    Hyp2[N0].Shade( SMOOTH, REFLECTING );
    Hyp2[N0].Contour( Black, MEDIUM );
    Hyp2[N0].ULines( Black, 2, MEDIUM );
    Hyp2[N0].Rotate( Xaxis, 180 );
}
```

♢

Example 3.15. Self-intersections
Now to the self-intersection. In the original book ([14]), we introduced the Klein bottle, which is an example of a one-sided surface (another is the MÖBIUS strip). By definition, a Klein bottle must have a self-intersection. When you display it and you show parameter lines and the contour, but you do not show the self-intersection, there is something important missing!

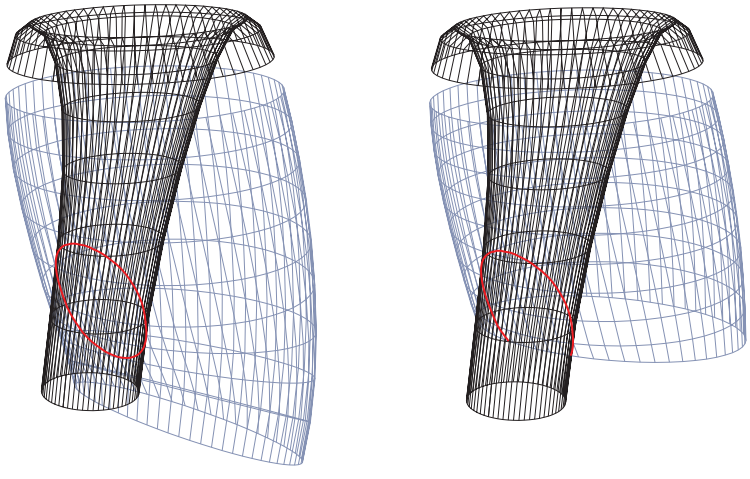

FIGURE 3.21. How to split the Klein bottle in order to get the self-intersection.

The trick to get the self-intersection is to split the surface into two parts. In our case, it turned out to be a good idea to split the u-interval of the surface ($u \in [0, 2\pi]$) into two intervals $u \in [2.5, 4]$ and $u \in [4.2, 6.5]$ (Figure 3.21), left). The rule is that the intervals must not overlap. They do not have to cover the entire parameter range; quite the contrary: The intervals should be as small as possible!

You will agree that this needs testing. For exammple, if we had chosen the slightly different intervals $u \in [3, 4]$ and $u \in [4.2, 6.5]$, the result would have been only part of the intersection curve (Figure 3.21, right). Therefore, for the time being we do the calculation of self-intersection in the drawing part of the scene:

Listing from `"klein_bottle.cpp"`:

```
ComplexL3d SelfIntersection;
...
void Scene::Draw( )
{
    Bottle.GetSelfIntersection( SelfIntersection,
        'u', 2.5, 4, 4.2, 6.5, 10, true );
    return;
```

```
  ...
}
```

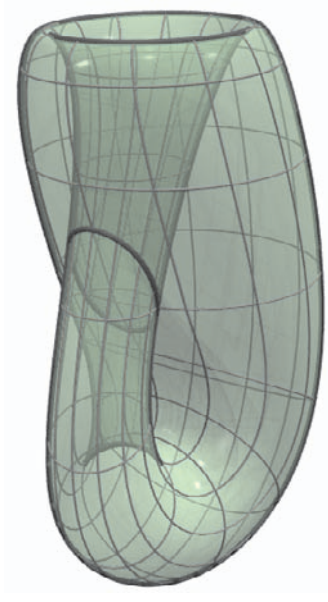

FIGURE 3.22. The famous Klein bottle has by definition a self-intersection (output of "klein_bottle.cpp").

The second parameter of GetSelfIntersection(...) tells the program that the u-interval has to be split (the only alternative is 'v'); the following four parameters are the borders of the two intervals, the next parameter is an "accuracy parameter." In our case, the relatively small number 10 was chosen (20 is the default; the number may be as high as 200, but this will consume time, and in most cases, it is not necessary). The last parameter, finally, tells the program to display a (two-colored) wire frame, including the intersection line (Figure 3.21). But keep in mind that drawing is done only in the actual drawing part!

When everything is fine, move the line with the GetSelfIntersection(...) command into to the initializing part and remove the **return** statement. The output is now worth seeing (Figure 3.22)!

Another example of a self-intersection is the snail in Figure 3.23. In this case, the u-interval was split by the line

```
Surface.GetSelfIntersection( Section, 'u', -PI, 0.8, 0.8 + 0.01,
                                                        PI, 20, true );
```

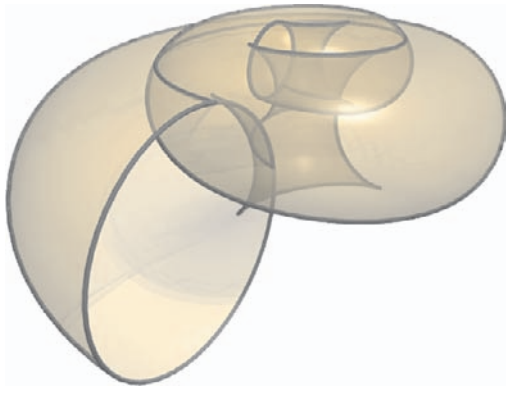

FIGURE 3.23. The shell of the snail — a spiral surface — has two branches of self-intersections (output of `snail_with_self_intersection.cpp`).

As a final example, we display a minimal surface, one of the surfaces in between the catenoid and the helicoid. (A soap bubble forms this surface when a wire frame is dipped into soapy water.) Figure 3.24 shows two views of the surface, and the wire frame in the testing phase, created by the line

```
Surf.GetSelfIntersection( Section, 'u',
        -1.2, 0.3, 0.3 + 0.01, 1.2, 12, true );
```

♢

Surface with thickness

OPEN GEOMETRY 2.0 provides a new class called *SurfWithThickness*. A parameterized surface Φ, an offset surface Ω of Φ, and a transition surface T, connecting the borderlines of Φ and Ω, are displayed. This creates images of high optical appeal and supports our 3D perception of the displayed objects.

Example 3.16. Cage of gold
The geometry of the sample program `"cage_of_gold1.cpp"` is very simple. We want to display some circles of longitude and latitude on a sphere Σ. They are realized as 3D surfaces with thickness. Circles of longitude and latitude will touch each other and thus create the impression of solid objects.

As usual, we define some global constants to determine the radius of Σ, the offset distance, and the number of circles. Then we have to write a function returning a surface point of Σ:

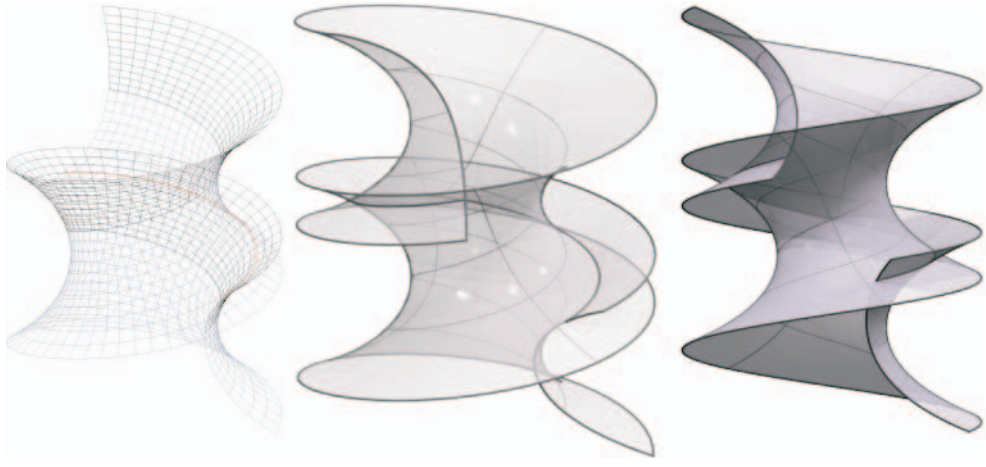

FIGURE 3.24. A minimal surface with self-intersections (output of `"minimal.cpp"`).

Listing from `"cage_of_gold1.cpp"`:

```
P3d PointOnSphere( Real u, Real v )
{
    P3d P( 0, Radius, 0 );
    P.Rotate( Xaxis, Deg( u ) );
    P.Rotate( Zaxis, Deg( v ) );
    return P;
}
```

Next, we define global variables of type *SurfWithThickness* (to be precise: pointers to objects of type *SurfWithThickness*), allocate the memory, and define the surfaces:

Listing from `"cage_of_gold1.cpp"`:

```
// 5 circles of latitude, N circles of longitude
SurfWithThickness *CircOfLat[5];
SurfWithThickness *CircOfLong[N];

void Scene::Init( )
{
    // Allocate memory for the surfaces.
    int i;
    for ( i = 0; i < 5; i++ )
        CircOfLat[i] = new SurfWithThickness( PointOnSphere );
```

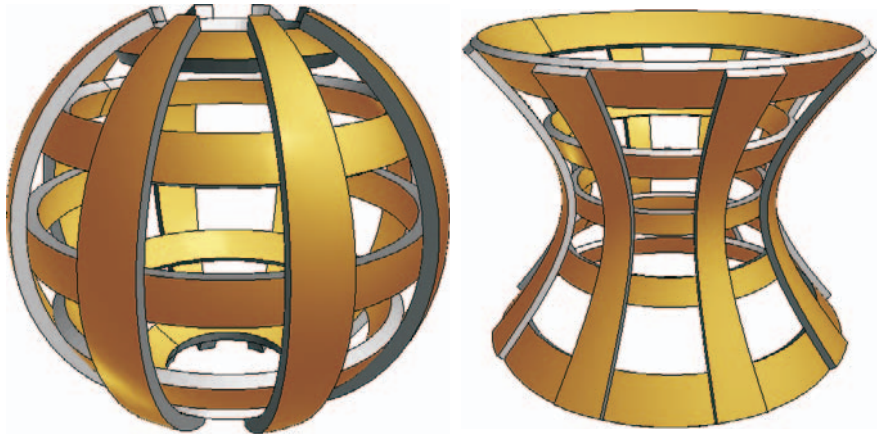

FIGURE 3.25. A spherical and a hyperbolic cage of gold.

```
    for ( i = 0; i < N; i++ )
        CircOfLong[i] = new SurfWithThickness( PointOnSphere );

    // Parameter limits for circles of latitude.
    Real u0 = 1.0, u1 = 1.2;
    Real u2 = -0.5 * ( u1 - u0 ), u3 = -u2;
    Real u4 = 0.5 * ( u1 + u2 ) + 0.5 * ( u1 - u0 );
    Real u5 = 0.5 * ( u1 + u2 ) - 0.5 * ( u1 - u0 );

    // Parameter limits for circles of longitude.
    Real v0 = 0, v1 = 0.25;

    // Define circles of latitude.
    CircOfLat[0]->Def( Yellow, 4, 75, u0, u1, -PI, PI );
    CircOfLat[1]->Def( Yellow, 4, 75, -u1, -u0, -PI, PI );
    CircOfLat[2]->Def( Yellow, 4, 75, u2, u3, -PI, PI );
    CircOfLat[3]->Def( Yellow, 4, 75, u4, u5, -PI, PI );
    CircOfLat[4]->Def( Yellow, 4, 75, -u5, -u4, -PI, PI );

    // Define circles of longitude.
    Real delta = 2 * PI / N;
    for ( i = 0; i < N; i++ )
        CircOfLong[i]->Def( Yellow, 25, 4, -u1, u1,
                            v0 + i * delta, v1 + i * delta );
}
```

Note that this procedure is essentially different from the implementation of a parameterized surface (**class** *ParamSurface*). In Init( ) we allocate the dynamic memory using the keyword **new**.[12] For the definition of the surfaces we use the **this** pointer −> and the usual input parameters (color, number of segments in the u- and v-directions and u- and v-parameter range).

In our example we set the reals u0,...,u5 in a way to achieve a good distribution of five circles of latitude. The N circles of longitude are equally distributed along the sphere. The Draw( ) part is very simple: We simply shade the surfaces.

Listing from "cage_of_gold1.cpp":

```
void Scene::Draw( )
{
   int i;
   for ( i = 0; i < 5; i++ )
      CircOfLat[i]->ShadeWithThickness( DarkGray,
         Thickness, THIN );
   for ( i = 0; i < N; i++ )
      CircOfLong[i]->ShadeWithThickness( DarkGray,
         -Thickness, THIN );
}
```

The additional input parameters specify the color of the small transition surface, the offset distance, and the linestyle of the outline. An optional argument of type *Boolean* decides whether the contour outline will be displayed as well. Finally, do not forget to free the dynamically allocated memory in CleanUp( )!

Listing from "cage_of_gold1.cpp":

```
void Scene::CleanUp( )
{
   int i;
   for ( i = 0; i < 5; i++ )
      delete CircOfLat[i];
   for ( i = 0; i < N; i++ )
      delete CircOfLong[i];
}
```

[12] If you are not familiar with dynamic memory allocation, we recommend using the Open Geometry macros ALLOC_ARRAY and FREE_ARRAY (compare page 598).

The output can be seen in Figure 3.25. There we display a little variation of the cage of gold as well (`"cage_of_gold2.cpp"`). We simply replaced the sphere Σ by a hyperboloid H of revolution. ♢

Further examples

In the remaining part of this chapter we give examples of two more fascinating surface types: PLÜCKER's conoid and supercyclides.

Example 3.17. A host of Plücker conoids
One of the most interesting geometric surfaces is Plücker's conoid Π. Generations of mathematicians have been fascinated by Π since its discovery by J. PLÜCKER in 1868. We will illustrate a few of their results in a sequence of programs.

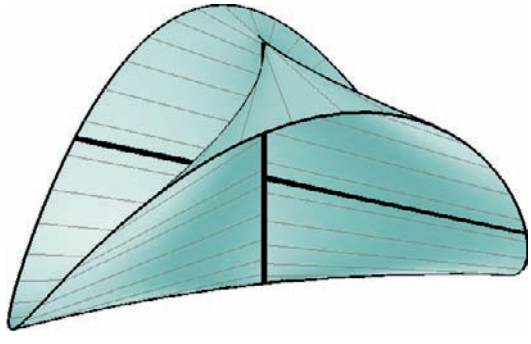

FIGURE 3.26. Plücker's conoid Π.

To start with, we use a parameterized equation to produce a picture of Π. An implicit equation of Π is

$$\Pi \ldots z(x^2 + y^2) - hxy = 0. \tag{1}$$

The real number h is called the height of Π. We can rewrite this equation in cylindrical coordinates as

$$\Pi \ldots z = \frac{h}{2} \sin 2\varphi.$$

This gives rise to a first geometric definition of Π:

Let g be a straight line intersecting the z-axis perpendicularly. Rotate g with constant angular velocity about z and let it swing harmonically along z. One full rotation corresponds to two swinging periods. Then g sweeps Plücker's conoid Π.

There are many other ways of defining Π. We will see some of them in the following text. But they are not so convenient for obtaining a good parameterization of Π.

In `"pluecker1.cpp"` we draw a nice image of Π. We implement Π as a ruled surface according to our considerations:

Listing from `"pluecker1.cpp"`:

```
class MyPlueckerConoid: public RuledSurface
{
   virtual P3d DirectrixPoint( Real u )
   {
      return P3d( 0, 0, 0.5 * Height * sin( 2 * u ) );
   }
   virtual V3d DirectionVector( Real u )
   {
      return V3d( cos( u ), sin( u ), 0 );
   }
};
MyPlueckerConoid PlueckerConoid;
```

The remaining drawing is OPEN GEOMETRY standard and we omit displaying it here. Using the Generator(...) method of *RuledSurface* we draw a rotating and swinging generator of Π:

Listing from `"pluecker1.cpp"`:

```
StrL3d g;
g = PlueckerConoid.Generator( U );
g.Draw( Black, V0, V1, THICK, 1e-3 );
```

Here, U, V0, and V1 are global variables, U is increased in Animate( ); $[V0, V1]$ is the v-parameter range of our surface.

Now we proceed a little further. On Π we find a two-parameter manifold of conic sections. They are all ellipses that intersect the z-axis. Their top view is always a circle. These considerations immediately result in a second definition of Π:

Let ζ be a cylinder of revolution and let e and g be an ellipse and a straight line, respectively, on ζ. Then Π *consists of all horizontal lines that intersect e and g.*

We illustrate this in `"pluecker2.cpp"`. To begin with, we create Π exactly as in `"pluecker1.cpp"`. It is not a good idea to use the second definition of Π to derive a parametric representation. This would inevitably result in numerical problems with direction vectors almost identical to the zero vector.

Then we write a function taking two input parameters x and y and returning the point $P(x, y, z) \in \Pi$ according to equation (1). The point P is not uniquely determined iff $x = y = 0$. In this case we simply return the origin.

Listing from "pluecker2.cpp":

```
P3d PointOnConoid( Real x, Real y )
{
    if ( x == 0 && y == 0 )
        return Origin;
    else
        return P3d( x, y, Height * x * y / ( x * x + y * y ) );
}
```

Now we take a cylinder of revolution ζ through z and use the function PointOnConoid(...) to parameterize the part below the elliptical intersection with Π.

Listing from "pluecker2.cpp":

```
class MyCylinder: public ParamSurface
{
public:
    virtual P3d SurfacePoint( Real u, Real v )
    {
        P3d M( 2, 5, -0.5 * Height );
        P3d M1( 2 * M.x, 2 * M.y, 0 );
        P3d N1( M.x - M.y, M.x + M.y, 0 );
        P3d O1( M.x + M.y, M.y - M.x, 0 );
        M1 = PointOnConoid( M1.x, M1.y );
        N1 = PointOnConoid( N1.x, N1.y );
        O1 = PointOnConoid( O1.x, O1.y );
        Plane p;
        p.Def( M1, N1, O1 );
        Real rad = sqrt( M.x * M.x + M.y * M.y );
        P3d P( M.x + rad * cos( u ), M.y + rad * sin( u ),
               -0.5 * Height );
        StrL3d s( P, Zdir );
        P3d Q = p * s;
        return ( 1 - v ) * P + v * Q;
    }
};
MyCylinder Cylinder;
```

We specify the midpoint of ζ, take three points M1, N1, and O1 on a base circle, and project them onto the conoid by means of PointOnConoid(...). These points determine a plane π. We parameterize the circle with z-coordinate $-h/2$ and take the corresponding point in π to get the desired parametric representation.

You would probably like to know why we use the auxiliary plane π instead of the PointOnConoid(...) function to find the points on the ellipse. The reason is simple: PointOnConoid(...) is numerically unstable in the neighborhood of the origin. Try it and watch the ugly cylinder you get!

The rest is simple. We shade Π and ζ and draw a generator of Π that runs across the surface during the animation. In Draw( ) we write the following lines:

Listing from `"pluecker2.cpp"`:

```
P3d A = Cylinder.SurfacePoint( U, 1 );
P3d B = Zaxis.NormalProjectionOfPoint( A );
StraightLine3d( Black, A, B, THICK, 0.005 );
A.Mark( Black, 0.3, 0.2 );
B.Mark( Black, 0.3, 0.2 );
```

Here, U is a global variable of type *Real* that is increased in Animate( ). The output can be seen in Figure 3.27.

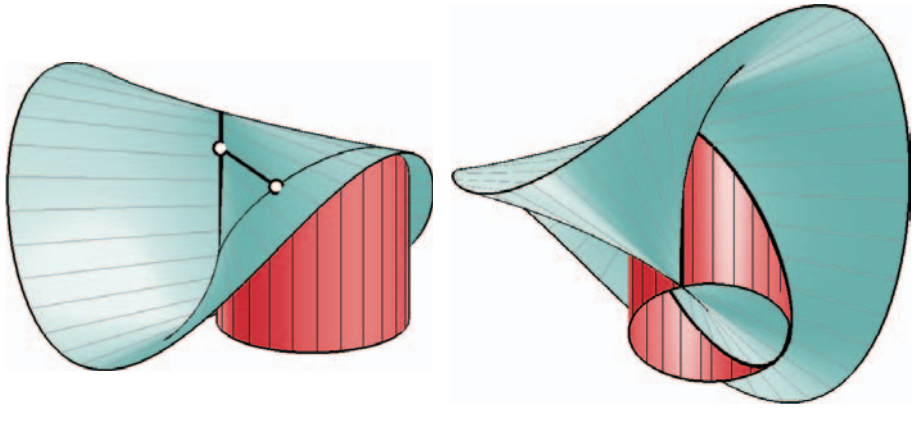

FIGURE 3.27. Alternative generation of Plücker's conoid: Any cylinder of revolution through the axis intersects Π in an ellipse.

Of course, it is interesting to visualize the different ellipses on Π. We do this in `"pluecker3.cpp"`. There are only few differences between this file and `"pluecker2.cpp"`. Firstly, we use two global real variables X and Y as coordinates for the base circle's center.

Listing from `"pluecker3.cpp"`:

```
class MyCylinder: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real u, Real v )
   {
      P3d M( X, Y, -0.5 * Height );
      ...
   }
```

Secondly, we do not compute the surface points of the cylinder in Init( ) but in Draw( ) (PulsingX and PulsingY are two pulsing reals):

Listing from `"pluecker3.cpp"`:

```
X = PulsingX.Next( );
Y = PulsingY.Next( );
Cylinder.Def( Red, 50, 5, -PI, PI, 0, 1 );
```

And thirdly, we no longer animate a generator of Π. Only the cylinder of revolution changes from frame to frame.

There exists a very interesting kinematical way to create Plücker's conoid Π. Suppose that ζ and Z are both cylinders of revolution with radii r and $R = 2r$, respectively. Then we have the following theorem:

If ζ rolls inside Z, an arbitrary ellipse e of ζ sweeps Plücker's conoid.

In `"pluecker4.cpp"` we define and shade Π as usual. Additionally, we need a few global variables and constants:

Listing from `"pluecker4.cpp"`:

```
const Real BigRadius = 12;
const Real SmallRadius = 0.5 * BigRadius;
const Real Phi = 1;
const Real Psi = -2 * Phi;
StrL3d RollingAxis;
L3d RollingEllipse;
const int N = 18;
L3d FixedEllipse[N];
```

They define the radii of the fixed and the rolling cylinder two angle increments for the simulation of the rolling motion, the axis of the rolling cylinder, the ellipse on the rolling cylinder and a number N of ellipses to be drawn on Π. Then we define the fixed cylinder:

Listing from `"pluecker4.cpp"`:

```
class MyFixedCylinder: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real u, Real v )
   {
      const Real radius = BigRadius + 0.05;
      P3d P( radius * cos( u ), radius * sin( u ), -0.5 * Height );
      P3d Q = PointOnConoid( P.x, P.y );
      return ( 1 - v ) * P + v * Q;
   }
};
MyFixedCylinder FixedCylinder1, FixedCylinder2;
```

Its radius is a little larger than BigRadius in order to avoid visibility problems (the two cylinders actually touch each other along a generator). Then we parameterize the base circle and determine the corresponding points on Π. The line segment between $P(u)$ and $Q(u)$ will have just the right length. Note that we create two instances of the fixed cylinder. We will use the second instance to create a complement of the lower cylinder part (compare Figure 3.28).

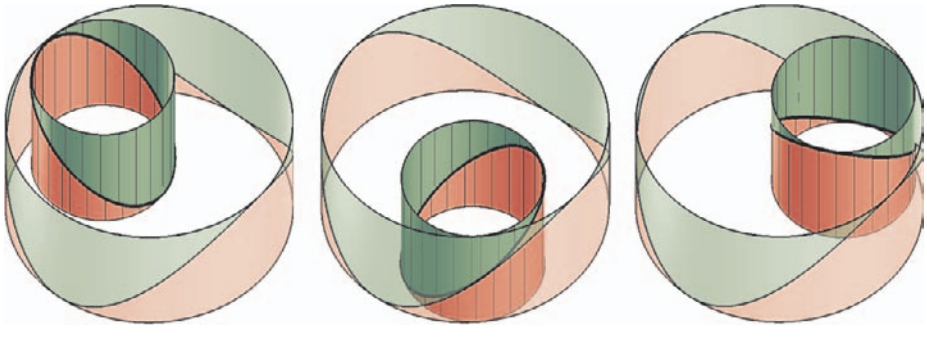

FIGURE 3.28. A cylinder of radius r rolls inside a cylinder of radius $R = 2r$.

The rolling cylinder is defined similarly to the previous programs:

Listing from "pluecker4.cpp":

```
class MyRollingCylinder: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real u, Real v )
   {
      P3d M( SmallRadius, 0, -0.5 * Height );
      P3d M1( 2 * M.x, 2 * M.y, 0 );
      P3d N1( M.x - M.y, M.x + M.y, 0 );
      P3d O1( M.x + M.y, M.y - M.x, 0 );
      M1 = PointOnConoid( M1.x, M1.y );
      N1 = PointOnConoid( N1.x, N1.y );
      O1 = PointOnConoid( O1.x, O1.y );
      Plane pi;
      pi.Def( M1, N1, O1 );
      Real rad = sqrt( M.x * M.x + M.y * M.y );
      P3d P( M.x + rad * cos( u ), M.y + rad * sin( u ),
            -0.5 * Height );
      StrL3d s( P, Zdir );
      P3d Q = pi * s;
      return ( 1 - v ) * P + v * Q;
   }
};
MyRollingCylinder RollingCylinder1, RollingCylinder2;
```

Again we will use two instances of the class. The Init( )part turns out to be of more interest than Draw( ). We give a complete listing

Listing from "pluecker4.cpp":

```
void Scene::Init( )
{
   PlueckerConoid.Def( LightBlue, 50, 50, 0, PI,
                       -BigRadius, BigRadius );
   PlueckerConoid.PrepareContour( );

   FixedCylinder1.Def( Red, 80, 5, -PI, PI, 0, 1 );
   FixedCylinder1.PrepareContour( );

   FixedCylinder2.Def( Green, 80, 5, -PI, PI, 0, 1 );
   FixedCylinder2.Reflect( XYplane );
   FixedCylinder2.Rotate( Zaxis, 90 );
   FixedCylinder2.PrepareContour( );

   RollingAxis.Def( P3d( SmallRadius, 0, 0 ), Zdir );
```

```
    RollingCylinder1.Def( Red, 50, 5, −PI, PI, 0, 1 );

    RollingCylinder2.Def( Green, 50, 5, −PI, PI, 0, 1 );
    RollingCylinder2.Reflect( XYplane );
    RollingCylinder2.Rotate( RollingAxis, 180 );

    RollingCylinder1.GetULine( RollingEllipse, 1, DarkBlue );

    Real delta = (Real) 360 / N;
    int i;
    for ( i = 0; i < N; i++ )
    {
      FixedEllipse[i] = RollingEllipse;
      FixedEllipse[i].Rotate( Zaxis, ( i + 1 ) * delta );
      RollingAxis.Rotate( Zaxis, delta );
      FixedEllipse[i].Rotate( RollingAxis, − 2 * ( i + 1 ) * delta );
    }
    RollingEllipse.ChangeColor( Black );
  }
```

We define the conoid and the fixed cylinders. The second instance of the fixed cylinder is transformed to a complement of the first. Both parts together yield just an ordinary cylinder of revolution. This is possible because of certain symmetries of Π.

Next, we define the axis of the rolling cylinders and the rolling cylinders themselves. One borderline of the first instance is the rolling ellipse. We use it to define the ellipses on Π in the for-loop. Note that the rolling axis is at the same position before and after the loop.

In Draw( ) we just draw and shade the relevant elements (the two parts of the fixed cylinder are displayed with transparency). It is not necessary to list it here. In Animate( ) we revolve the rotating cylinder's parts, the ellipse on it, and its axis (the rotating axis) about z through the angle Phi. The first elements are then revolved about the rotating axis through the angle $\Psi = -2\text{Phi}$. This gives exactly the rolling motion. The output of the program is displayed in Figure 3.29.

Listing from `"pluecker4.cpp"`:

```
void Scene::Animate( )
{
  RollingCylinder1.Rotate( Zaxis, Phi );
  RollingCylinder2.Rotate( Zaxis, Phi );
  RollingEllipse.Rotate( Zaxis, Phi );
  RollingAxis.Rotate( Zaxis, Phi );
```

```
    RollingCylinder1.Rotate( RollingAxis, Psi );
    RollingCylinder2.Rotate( RollingAxis, Psi );
    RollingEllipse.Rotate( RollingAxis, Psi );
}
```

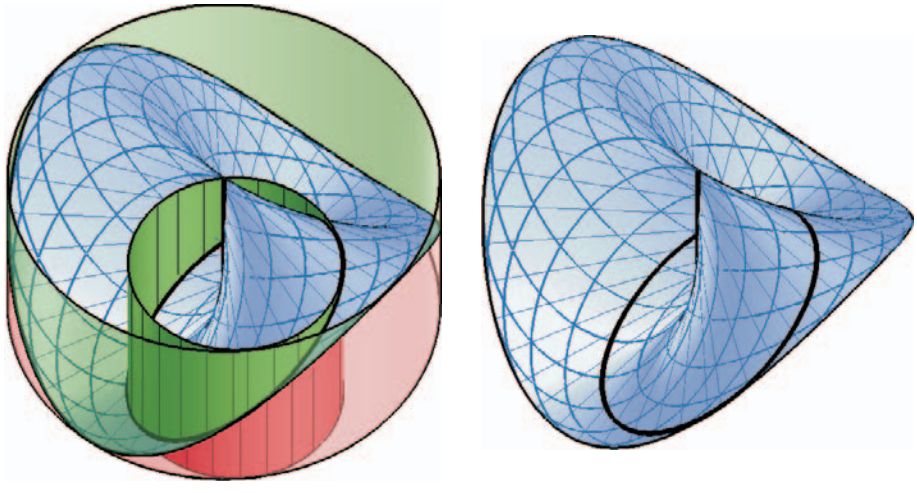

FIGURE 3.29. The ellipse on the rolling cylinder sweeps Plücker's conoid Π.

Closely related to the above definitions of Π is the following: *The common normals of a straight line d and the members of a general pencil of lines (apex* E*, supporting plane ε) generate Plücker's conoid.*

We illustrate this in `"pluecker5.cpp"`. But before we start, we have to take some things into consideration. What is the connection between d, E, ε, and the conoid Π? For reasons of symmetry d has to be the double line of Π. As before, we will use it as the z-axis of our coordinate system. Now everything will become clear if we consider the top view. The feet of the common normals seemingly lie on a circle with diameter $z'\mathsf{E}'$. Thus, the current generation of Π is a disguised version of the generation we used in `"pluecker2.cpp"`.

The plane ε has to be the supporting plane of an ellipse $e \subset \Pi$ or, since Π is of order 3, a tangent plane of Π. Finally E is an arbitrary point on e. This is all we have to know for our program.

In addition, to shade our standard conoid Π, we implement an ellipse on Π as a parameterized curve:

Listing from `"pluecker5.cpp"`:

```
class MyConoidEllipse: public ParamCurve3d
```

```
{
public:
   P3d CurvePoint( Real u )
   {
      const P2d M( 2, −4 );
      const Real radius = sqrt( M.x ∗ M.x + M.y ∗ M.y );
      P2d P( M.x + radius ∗ cos( u ), M.y + radius ∗ sin( u ) );
      return PointOnConoid( P.x, P.y );
   }
};
MyConoidEllipse ConoidEllipse;
```

We choose a curve point E and compute the normal of the ellipse's supporting plane ε in Init( ):

Listing from "pluecker5.cpp":

```
P3d A = ConoidEllipse.CurvePoint( 1.1974 );
P3d B = ConoidEllipse.CurvePoint( 3.7479 );
V3d normal = V3d( A, B ) ^ V3d( A, E );
Axis.Def( E, normal );
```

In Draw( ), we take a curve point P of the ellipse and its normal projection Q on the z-axis. The straight line [P, Q] is a generator of Π. Varying P yields the animation (Figure 3.30).

Next, we draw the relevant lines and points. In order to create a nice image, we draw a circle in ε around E. The plane ε intersects Π in the ellipse and another straight line s. We draw this line as well:

Listing from "pluecker5.cpp":

```
void Scene::Draw( )
{
   // Draw the axis of the conoid.
   Zaxis.Draw( Green, −0.5 ∗ Height, 0.5 ∗ Height, THICK, 1e−2 );

   // Shade the conoid.
   PlueckerConoid.Shade( SMOOTH, REFLECTING );
   PlueckerConoid.VLines( DarkBlue, 20, THIN );
   PlueckerConoid.ULines( Black, 2, MEDIUM );
   PlueckerConoid.Contour( Black, MEDIUM );

   ConoidEllipse.Draw( THICK, 1e−3 );

   // A point on the ellipse...
```

```
    P3d P = ConoidEllipse.CurvePoint( U );
    // ...and its corresponding point on z.
    P3d Q = Zaxis.NormalProjectionOfPoint( P );

    StrL3d s;
    s.Def( E, P );
    s.Draw( Red, -Radius, Radius, THICK );
    StraightLine3d( Green, P, Q, THICK );

    E.Mark( Red, 0.2, 0.1 );
    P.Mark( Black, 0.2, 0.1 );
    Q.Mark( Green, 0.2, 0.1 );

    Circ3d circle;
    circle.Def( Orange, E, Axis.GetDir( ), Radius, 80, FILLED );

    Plane pi;
    pi.Def( E, Axis.GetDir( ) );
    P3d S, T;
    S = pi * Zaxis;
    pi.Def( S, Zdir );
    int n;
    n = circle.SectionWithPlane( pi, S, T );
    if ( n == 2 )
        StraightLine3d( Red, S, T, MEDIUM );

    circle.ShadeTransparent( Orange, 0.3, Red, MEDIUM );
}
```

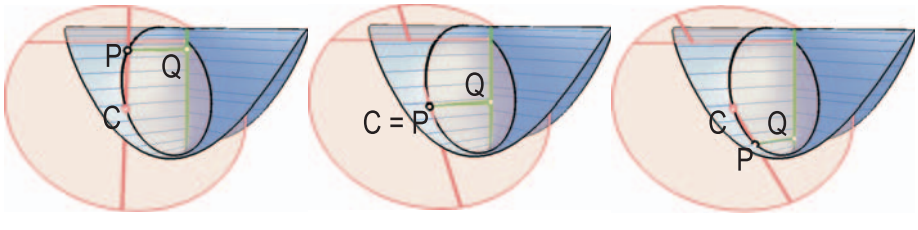

FIGURE 3.30. The common normals of a straight line and the members of a pencil of lines generate Π.

Another remarkable property of Π is the following: *All common normals of an arbitrary straight line s and the rulings on* Π *generate another example of Plücker's conoid.* If s is parallel to the axis of Π, the two conoids are even congruent.

Have a look at `"pluecker6.cpp"` if you don't believe it! We define the conoid Π_1 in almost the same way as in the previous examples. Its axis is parallel to z but does not contain the origin. The second conoid is defined in the geometric way we have just mentioned: We determine the direction vector of the common normal:

Listing from "pluecker6.cpp":

```
StrL3d Axis1( P3d( -V0, 0, 0 ), Zdir );
StrL3d Axis2( P3d(  V0, 0, 0 ), Axis1.GetDir( ) );

P3d PointOnAxis( Real u )
{
    return P3d( 0, 0, H2 * sin( 2 * u ) );
}

class MyPlueckerConoid1: public RuledSurface
{
public:
    virtual P3d DirectrixPoint( Real u )
    {
        return Axis1.NormalProjectionOfPoint( PointOnAxis( u ) );
    }
    virtual V3d DirectionVector( Real u )
    {
        return V3d( cos( u ), sin( u ), 0 );
    }
};
MyPlueckerConoid1 PlueckerConoid1;

class MyPlueckerConoid2: public RuledSurface
{
    virtual P3d DirectrixPoint( Real u )
    {
        return Axis2.NormalProjectionOfPoint( PointOnAxis( u ) );
    }
    virtual V3d DirectionVector( Real u )
    {
        return Axis2.GetDir( ) ^
               PlueckerConoid1.DirectionVector( u );
    }
};
MyPlueckerConoid2 PlueckerConoid2;
```

In addition, we use a *PathCurve3d* named Conic to display the path of the foot of the common normal on the generators of the conoids and an *L3d* object IntersectionCurve to mark the residual intersection curve. We define them in Init( ):

Listing from "pluecker6.cpp":

```
Conic.Def( Black, (int)(2 * PI / Phi) + 1 );

int n = 301;
IntersectionCurve.Def( Black, n );
Real u0 = -0.82, u1 = -u0;
Real delta = ( u1 - u0 ) / ( n - 1 );
int i;
Real u;
for ( i = 0, u = u0; i < n; i++, u += delta )
    IntersectionCurve[i] = YZplane *
        PlueckerConoid1.Generator( u );
```

There are just enough points on Conic to fill the whole elliptical intersection of the two conoids (Phi is the increment of the surface parameter u in Animate( )). The residual real and finite intersection consists of the x-axis and an algebraic curve of order three in $[y, z]$. We compute it by intersecting the generators of Π_1 with this plane. In Draw( ) we draw the conoids, determine the generating lines of a parameter value u, draw relevant lines, and mark relevant points (Figure 3.31).

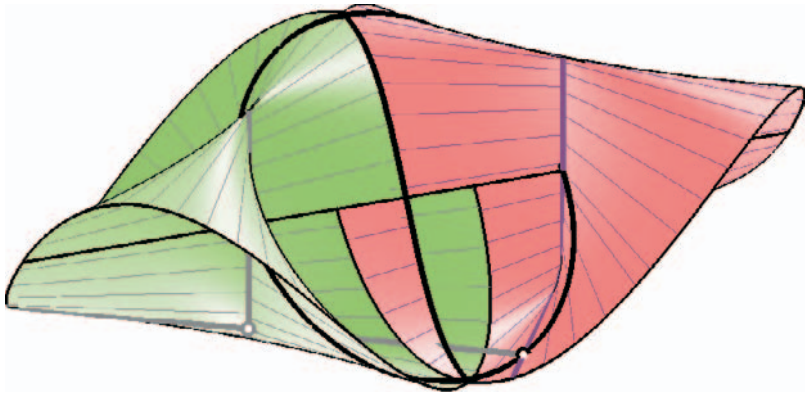

FIGURE 3.31. Two congruent Plücker conoids. Any generator of the first conoid intersects a generator of the second orthogonally in a common ellipse.

Given an arbitrary ruled surface Φ and a point P, we can construct the *pedal curve of* Φ *with respect to* P. The pedal curve consists of all normal projections of P on the generators of Φ. The class *RuledSurface* supports three methods concerning pedal curves:

- *P3d* PedalPoint(*Real* u, *P3d* Pole) returns the normal projection of *Pole* on the generator $g(\mathsf{u})$ of Φ.

- **void** GetPedalLine(*L3d* &pedal_line, *P3d* Pole, *Color* col, **int** size) stores the pedal line of Φ with respect to Pole in pedal_line.
- **void** DrawPedalLine(*P3d* Pole, *Color* col, int n, *ThinOrThick*, *Real* offset) draws the pedal line of Φ with respect to Pole.

We use this to illustrate yet another property of Plücker's conoid: It is the only nontrivial example of a ruled surface with *only plane pedal curves*. Have a look at the code of `"pluecker7.cpp"`. Besides minor things (drawing lines, marking points, etc.), the only relevant change to previous programs is in Draw( ). There we insert the following line:

Listing from `"pluecker7.cpp"`:

```
PlueckerConoid.DrawPedalLine( Pole, Red, 150, THICK, 1e-2 );
```

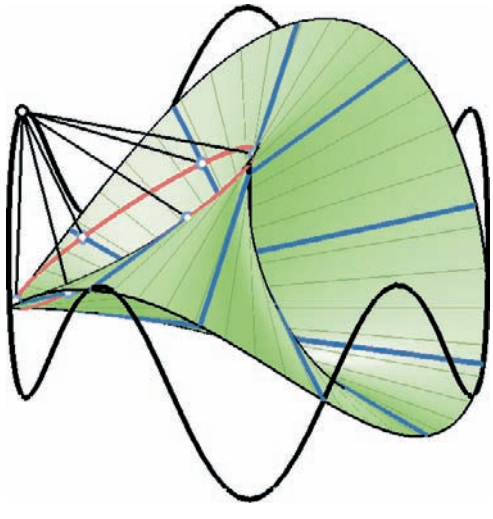

FIGURE 3.32. The pedal curves of Π are ellipses ("pluecker7.cpp").

Varying the global variable *P3d* Pole yields a one-parameter family of plane pedal curves. You will immediately note that they are just congruent ellipses on Π. In fact, the top view of the pedal curve with respect to a pole P is a circle with diameter OP'. In our program P' has always the same distance from O, so all pedal curves are congruent. ♢

Example 3.18. Supercyclides
An interesting class of surfaces are *supercyclides*. They are very special surfaces of conic sections. On a supercyclide Σ we find two families $\mathcal{F}_1$, $\mathcal{F}_2$ of conics. The planes of the conics of $\mathcal{F}_i$ belong to a pencil of planes with axis a_i (*characteristic lines*). The tangent planes of Σ along a conic $c_1 \in \mathcal{F}_1$ envelop a quadratic cone with apex on a_2 and the other way round. There exist quite a few different types of supercyclides. For example, one can distinguish them with respect to the position of the characteristic lines. Originally dating back to the nineteenth century ([3]), supercyclides turned up only recently in CAGD (see, e.g., [28, 29]).

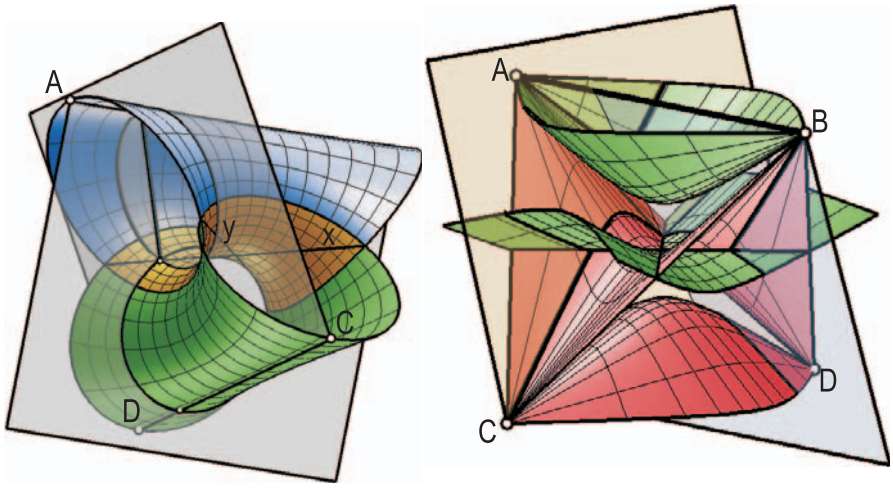

FIGURE 3.33. Two supercyclides.

In the following we will give examples of different types of supercyclides and visualize some of their geometric properties. In order to do so, we need a parametric representation. Note that the defining properties mentioned above are of a projective nature. We will therefore derive a projective parameterization at first and transform it to Euclidean coordinates in a second step. For those who are not so familiar with the basic concepts of projective geometry (especially homogeneous coordinates) we recommend a quick glance at the corresponding section in Chapter 5.

Let A, B, C, and D be four independent points and $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$ their homogeneous coordinate vectors. With the help of three independent quadratic polynomials $f(s)$, $a(s)$, and $b(s)$ in the variable s and three independent quadratic polynomials $g(t)$, $c(t)$, and $d(t)$ in the variable t, a homogeneous parametric representation of Σ reads

$$\Sigma \ldots \mathbf{x}(s,t) = g(t)[a(s)\mathbf{a} + b(s)\mathbf{b}] + f(s)[c(s)\mathbf{c} + d(s)\mathbf{d}].$$

The parameter range for s and t is $\mathbb{R} \cup \infty$. For example $\mathbf{x}(\infty, t)$ is defined as $\lim_{s\to\infty} s^{-2}\mathbf{x}(s,t)$. It is always a little dangerous to use homogeneous parametric

representations for the visualization of surfaces. The zeros of the x_0-coordinate of $\mathbf{x}(s,t)$ yield points at infinity, which are not really good for a triangulation (i.e., you cannot use them at all). Already parameter values that are too close to the zeros of the x_0-coordinate can have disastrous effects on the image. We will have to restrict the parameters to intervals that avoid these zeros.

The first task we set ourselves is the development of a useful OPEN GEOMETRY mask for the visualization of supercyclides. It should be possible to produce an attractive image by changing the relevant input data without too much effort. You should always try to do this when you have to write a series of similar programs (compare Section 7.6).

Now consider the program `"supercyclide1.cpp"`. There, we start by defining the points A, B, C, and D. It will be a good idea to give them "nice" (i.e., symmetric) positions in the Euclidean sense. They are situated at the corners of a cube centered at the origin in a way that AB and CD are skew but form a right angle.

Listing from `"supercyclide1.cpp"`:

```
const Real K = 8;
const P3d PA( -K, -K, -K ), PB(  K, K, -K );
const P3d PC(  K, -K,  K ), PD( -K, K,  K );
```

Now we have to define the quadratic forms $f(s)$, $a(s)$, and $b(s)$. In our example we choose $f(s) = s^2 + 1$, $a(s) = s(s+1)$, and $b(s) = s(s-1)$.

Listing from `"supercyclide1.cpp"`:

```
const Real F0 = 1, F1 =  0, F2 = 1;
const Real A0 = 0, A1 =  1, A2 = 1;
const Real B0 = 0, B1 = -1, B2 = 1;

Real F( Real s ) { return F0 + s * F1 + s * s * F2; }
Real A( Real s ) { return A0 + s * A1 + s * s * A2; }
Real B( Real s ) { return B0 + s * B1 + s * s * B2; }
```

The quadratic forms $g(t)$, $c(t)$, and $b(t)$ are defined in an analogous fashion. It is important to avoid getting confused by using homogeneous coordinates. Therefore, we write down everything in a neat and clean way. We start with the homogeneous coordinate functions:

Listing from "supercyclide1.cpp":

```
Real X0( Real s, Real t )
{
   return G( t ) * ( A( s ) * 1 + B( s ) * 1 ) +
          F( s ) * ( C( t ) * 1 + D( t ) * 1 );
}
Real X1( Real s, Real t )
{
   return G( t ) * ( A( s ) * PA.x + B( s ) * PB.x ) +
          F( s ) * ( C( t ) * PC.x + D( t ) * PD.x );
}
Real X2( Real s, Real t )
{
   return G( t ) * ( A( s ) * PA.y + B( s ) * PB.y ) +
          F( s ) * ( C( t ) * PC.y + D( t ) * PD.y );
}
Real X3( Real s, Real t )
{
   return G( t ) * ( A( s ) * PA.z + B( s ) * PB.z ) +
          F( s ) * ( C( t ) * PC.z + D( t ) * PD.z );
}
```

Note that we use the Euclidean coordinates of A, B, C, and D for the coordinate functions X1(s, t), X2(s, t), and X3(s, t), while X0(s, t) requires the coefficient 1 four times. We could, e.g., easily cast the point A to infinity if we replaced A(s)∗1 by A(s)∗0. Remember that we want to create a mask for later usage. Therefore, the redundancy of the above code seems to be justified. Now it is easy to get the parameterized equation of the supercyclide Σ:

Listing from "supercyclide1.cpp":

```
P3d CyclidPoint( Real s, Real t )
{
   P3d P( X1( s, t ), X2( s, t ), X3( s, t ) );
   P /= X0( s, t );
   return P;
}
```

You may wonder why we do not implement a parameterized surface at once. It is again for reasons of readability. We cannot exhaust the whole parameter interval $\mathbb{R} \cup \infty$. Hence, we have to split the surface and parameterize each part separately. It is a question of trial and error (or intuition) how many patches are necessary and how they are to be parameterized. We cannot give a general rule. You will have to try on your own. For supercyclides it seems to be a sensible idea to start with four different patches Σ_i with parameterizations

$$\mathsf{X1}(\mathsf{s},\mathsf{t}), \quad \mathsf{X2}(1/\mathsf{s},\mathsf{t}), \quad \mathsf{X3}(\mathsf{s},1/\mathsf{t}), \quad \text{and} \quad \mathsf{X4}(1/\mathsf{s},1/\mathsf{t}),$$

where s and t range over $(-1,1)$. You probably have to change the parameter range for one patch or the other. Perhaps, one or two of them cannot be used at all because of "infinity problems." The corresponding OPEN GEOMETRY code for Σ_1 and Σ_2 is as follows:

Listing from `"supercyclide1.cpp"`:

```
class MySuperCyclide1: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real s, Real t )
   {
      return CyclidPoint( s, t );
   }
};
MySuperCyclide1 SuperCyclide1;

class MySuperCyclide2: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real s, Real t )
   {
      return CyclidPoint( 1 / s, t );
   }
};
MySuperCyclide2 SuperCyclide2;
```

In Init( ) we define all four patches. Additionally, we prepare the contour but only if a global constant of type *Boolean* is set to true. This is a convenient way to toggle between fast image generation and animation on the one hand and a high-quality output on the other.

We do the same with the drawing of the wire frame and the border lines of the surface patches. For the output in Figure 3.33 we set DrawWireLines and DrawBorderLines to **true**. Furthermore, we display only the three patches Σ_2, Σ_3, and Σ_4 for $\mathsf{s}, \mathsf{t} \in (0, 1)$.

In Figure 3.33 (left side) you can see that the conics on the surface lie in the planes of two pencils with axes $a_1 := \mathsf{AB}$ and $a_2 = \mathsf{CD}$. In order to produce this picture, we added a few more code lines (`"supercyclide2.cpp"`). There, we basically use the same code as in `"supercyclide1.cpp"` but add some additional features. We introduce the following global variables:

Listing from `"supercyclide2.cpp"`:

```
Poly3d Frame1, Frame2;
PulsingReal R;
Conic3d SLine, TLine;
P3d S[5], T[5];
```

and initialize some of them in Init( ):

Listing from `"supercyclide2.cpp"`:

```
const Real f = 0.9999;

const Real s01 = -0.6, s11 = 0.6;
const Real t01 = -0.5, t11 = 0.5;
SuperCyclide1.Def( Col[0], 20, 20, s01, s11, t01, t11 );

const Real s02 = -f, s12 = f;
const Real t02 = -0.7, t12 = 0.7;
SuperCyclide2.Def( Col[1], 20, 20, s02, s12, t02, t12 );

const Real s03 = -0.7, s13 = 0.7;
const Real t03 = -f, t13 = f;
SuperCyclide3.Def( Col[2], 20, 20, s03, s13, t03, t13 );

const Real s04 = -f, s14 = f;
const Real t04 = -f, t14 = f;
SuperCyclide4.Def( Col[3], 20, 20, s04, s14, t04, t14 );
```

One side of the two rectangles will always be a_1 and a_2, respectively. The pulsing real R will be used for their animation: Frame1 and Frame2 will rotate forward and backward about a_1 and a_2, respectively. They will intersect Σ in conics that will be stored in SLine and TLine (the conics are s- and t-parameter lines). We implement this in Draw( ):

Listing from "supercyclide2.cpp":

```
Real r = R( );
int i;
for ( i = 0; i < 5; i++ )
{
    S[i] = SuperCyclide3.SurfacePoint( i, r );
    T[i] = SuperCyclide2.SurfacePoint( r, i );
}
// SLine and TLine are instances of Conic3d!
SLine.Def( Black, 100, S );
TLine.Def( Black, 100, T );
SLine.Draw( THICK );
TLine.Draw( THICK );
S[0].Mark( Black, 0.2, 0.1 );
T[0].Mark( Black, 0.2, 0.1 );
P3d P = 0.5 * ( S[0] + T[0] );
P.Mark( Black, 0.2, 0.1 );
StraightLine3d( Black, S[0], T[0], THICK );

// bring the frames into correct position
// and shade with transparency
P.Def( 0, 0, -K );
V3d v = S[0] - P;
v.Normalize( );
Frame1[3] = P + 2.5 * K * v + V3d( K, K, 0 );
Frame1[4] = P + 2.5 * K * v - V3d( K, K, 0 );

P.Def( 0, 0, K );
v = T[0] - P;
v.Normalize( );
Frame2[3] = P + 2.5 * K * v - V3d( K, -K, 0 );
Frame2[4] = P + 2.5 * K * v + V3d( K, -K, 0 );

Frame1.ShadeTransparent( LightGray, 0.5, Black, MEDIUM );
Frame2.ShadeTransparent( LightGray, 0.5, Black, MEDIUM );
```

In a first step we initialize the conic points S[i], T[i] and define the corresponding conics. The points S[0] and T[0] play an important role: They are situated in planes of symmetry and can be used to adjust the frame rectangles. The midpoint P of S[0] and T[0] has a straight line x as trajectory during the animation. A second straight line y lies on Σ and is displayed in Figure 3.33 as well.

On the right-hand side of Figure 3.33 you see a different example of a supercyclide ("supercyclide3.cpp"). There, we used the polynomials

$$f(s) = 1 - s^2, \quad a(s) = 2s^2 - s - 1, \quad b(s) = 2s^2 + s - 1,$$
$$g(t) = 1 - t^2, \quad c(t) = 2t^2 - t - 1, \quad d(t) = 2t^2 + t - 1.$$

The parameter lines are hyperbolas, which makes the choice of the correct parameter intervals a little difficult. We split the surface into four patches and decide on the following code:

Listing from "supercyclide3.cpp":

```
const Real f = 0.9999;

const Real s01 = -0.6, s11 = 0.6;
const Real t01 = -0.5, t11 = 0.5;
SuperCyclide1.Def( Col[0], 20, 20, s01, s11, t01, t11 );

const Real s02 = -f, s12 = f;
const Real t02 = -0.7, t12 = 0.7;
SuperCyclide2.Def( Col[1], 20, 20, s02, s12, t02, t12 );

const Real s03 = -0.7, s13 = 0.7;
const Real t03 = -f, t13 = f;
SuperCyclide3.Def( Col[2], 20, 20, s03, s13, t03, t13 );

const Real s04 = -f, s14 = f;
const Real t04 = -f, t14 = f;
SuperCyclide4.Def( Col[3], 20, 20, s04, s14, t04, t14 );
```

Note that we limit certain parameter ranges by f = 0.9999. You may try what happens in the case of f = 1, but it will not be very attractive. In contrast to the previous example it is not a good idea to use the class *Conic3d* for the drawing of the surface conics: They are hyperbolas and will usually be drawn beyond the surface edges. We use globally declared objects of type *L3d* instead:

Listing from "supercyclide3.cpp":

```
Poly3d Frame1, Frame2;
PulsingReal R;
L3d SLine[4], TLine[4];
```

The corresponding code lines in Draw( ) have to be replaced by the following:

Listing from "supercyclide3.cpp":

```
Real r = R.Next( );
int n = 50;
int i;
SuperCyclide1.GetULine( SLine[0], -0.5 * r, Black, n );
SuperCyclide2.GetULine( SLine[1], -0.5 * r, Black, n );
SuperCyclide3.GetULine( SLine[2], r, Black, n );
SuperCyclide4.GetULine( SLine[3], r, Black, n );

SuperCyclide1.GetVLine( TLine[0], -0.5 * r, Black, n );
SuperCyclide2.GetVLine( TLine[1], r, Black, n );
SuperCyclide3.GetVLine( TLine[2], -0.5 * r, Black, n );
SuperCyclide4.GetVLine( TLine[3], r, Black, n );

SLine[0].Draw( THICK );
SLine[1].Draw( THICK );
SLine[2].Draw( THICK );
SLine[3].Draw( THICK );

TLine[0].Draw( THICK );
TLine[1].Draw( THICK );
TLine[2].Draw( THICK );
TLine[3].Draw( THICK );

StraightLine3d( Black,
                Axis2.NormalProjectionOfPoint( SLine[2][1] ),
                Axis1.NormalProjectionOfPoint( TLine[1][1] ),
                THIN );
```

That is, we draw objects of type *L3d* rather than objects of type *Conic3d*. The last line in the above listing draws the intersection line segment that lies in both frames and improves the 3D impression of the image. ♢

3.4 Modeling Surfaces

In the preceding chapter you got to know a number of parameterized surfaces from the field of pure geometry. In this chapter we will use those surfaces for the modeling of objects from everyday life. Of course, there exist standard techniques for that task (NURBS or Bézier spline surfaces; OPEN GEOMETRY 2.0 supports them as well). Sometimes, however, it could be necessary to develop shapes of your own in order to display a complex object. We will give you a few examples and, hopefully, useful hints to help you with this task.

Example 3.19. Energy spirals
In modeling a "real world" curve c or surface Φ it is sometimes more convenient to use *cylindrical coordinates* $(\varrho, \varphi, \zeta)$ instead of Cartesian coordinates (x, y, z). These two kinds of coordinates are linked by the equations

$$x = \varrho\cos(\varphi), \quad y = \varrho\sin\varphi, \quad z = \zeta. \tag{2}$$

The geometric meaning of ϱ, φ, and ζ is shown in Figure 3.34. This figure also explains the name of these coordinates: The point $P(\varrho, \varphi, \zeta)$ lies on a cylinder of revolution around the z-axis with radius ϱ.

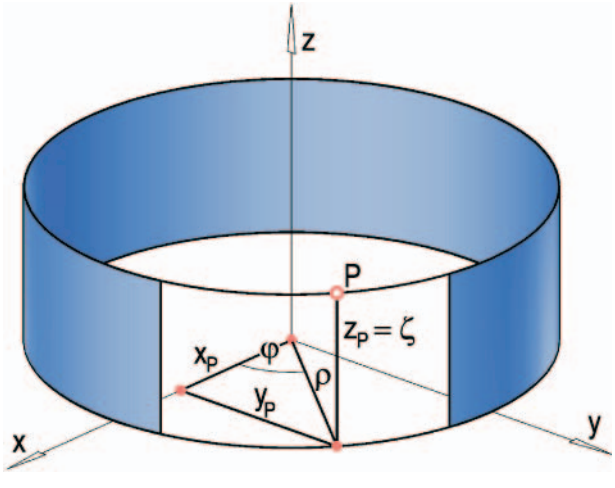

FIGURE 3.34. The geometric meaning of the cylindrical coordinates ϱ, φ, and ζ.

Though in principle, any curve or surface may be described in cylindrical coordinates, they are especially useful when some kind of rotational or helical symmetry with respect to the z-axis occurs.

As an example we present the model of a *Feng-Shui energy spiral* (see Figure 3.35). You can find those spirals in diverse shops nowadays. They are quite popular for their elegance: Suspending them on a string (the z-axis) and rotating them about z gives, depending on the sense of rotation, the impression of

an inside-up-outside-down or outside-down-inside-up motion. You can watch the same effect in our computer simulation (`"energy_spiral1.cpp"`).

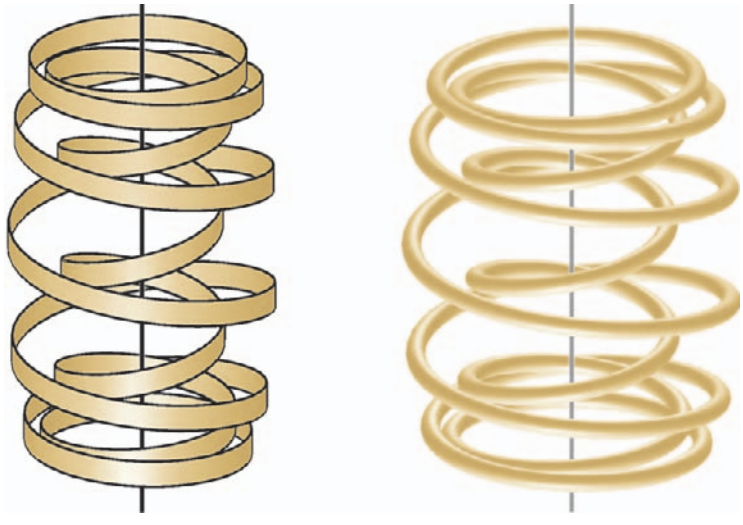

FIGURE 3.35. Two Feng-Shui energy spirals (the output of `"energy_spiral1.cpp"` and `"energy_spiral2.cpp"`).

The main programming problem is finding a closed curve c winding up along the z-axis in big loops and winding down again in small loops. We derive this curve from the well-known path curve of a point during a helical motion. In cylindrical coordinates its parametric representation simply reads

$$c' \dots C(\varphi) = (\varrho_0, \varphi, p\,\varphi)^T \tag{3}$$

with constant ϱ_0 and p (the screw parameter). Now, what is the difference between a helix and our target curve c? Obviously, the ϱ-coordinate of c cannot be constant. If we assume a parameter interval I that is symmetric with respect to 0, the graph of $\varrho(\varphi)$ must — more or less — look like the that drawn in Figure 3.36. The minimum value r and maximum value R of $\varrho(\varphi)$ in I are the minimum and maximum radii of the spiral surface. At the top and at the bottom of the spiral we get the radius $(R + r)/2$. The ζ-coordinate function must be a function symmetric with respect to the y-axis, looking like the graph on the right-hand side of Figure 3.36. Both functions have to be part of periodic C^m-functions (m being at least equal to one but preferably even bigger) with fundamental interval I in order to achieve appropriate smoothness of our model.

In our program we basically used the sine function:

$$\varrho(\varphi) = \frac{R-r}{2}\sin(n^{-1}\varphi) + \frac{R+r}{2}, \quad \zeta(\varphi) = \frac{h}{2}\sin(n^{-1}\varphi + \pi/2), \tag{4}$$

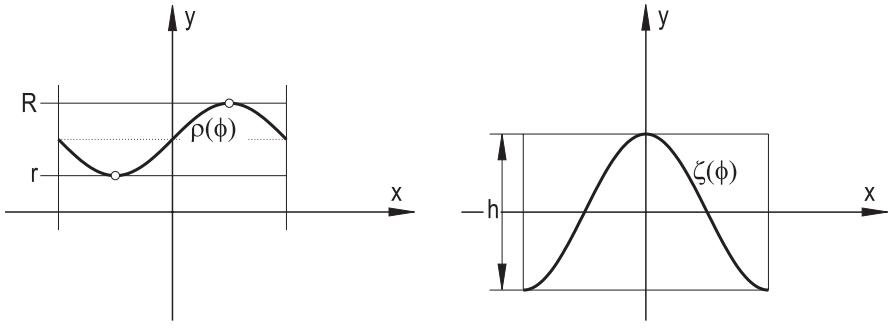

FIGURE 3.36. The two generating functions $\varrho(\varphi)$ and $\zeta(\varphi)$ of the energy spiral.

and $I = [-n\pi, n\pi]$. Here h denotes the total height of the surface and n the number of windings. According to (2) the final parametric representation of c in cylindrical coordinates is

$$c \ldots C(\varphi) = \begin{pmatrix} \varrho(\varphi)\cos(\varphi) \\ \varrho(\varphi)\sin(\varphi) \\ \zeta(\varphi) \end{pmatrix} \qquad \varphi \in [-n\pi, n\pi]. \tag{5}$$

It is now easy to obtain the parametric representation of the cylindrical spiral surface from our example. Switch on the auto-rotation about z to see the up-down movement! Alternatively, you can run `"energy_spiral2.cpp"`. There, the spiral is implemented as a tubular surface (Figure 3.35. ♢

Other examples of spiral surfaces can be found in modern architecture. For example, in the new Reichstag building in Berlin you can find a dome in the shape of an ellipsoid of revolution with two spirals winding up inside (Figure 3.37). Another example that was not realized but carefully planned (also from the static point of view!) can be seen on the right-hand side of Figure 3.37. It is a sculpture having the form of a reversed vortex; winding up outside and winding down inside, allowing people to walk up and down again without changing their direction.

Example 3.20. Candlestick
As our next example we present the computer visualization of an elegant candlestick (Figure 3.38). In the corresponding program `"candlestick.cpp"`, we will use different features of OPEN GEOMETRY such as parameterized curves and surfaces, surfaces of revolution, tubular surfaces, and the importation of CAD3D objects.

We divide the object into five parts. The top and the bottom parts are solids consisting of geometric primitives like spheres, cylinders, and cones of revolution. It is a good idea to model them in CAD3D (`"candlestick1.llz"` and `"candlestick2.llz"`) and import them in OPEN GEOMETRY. We define a global

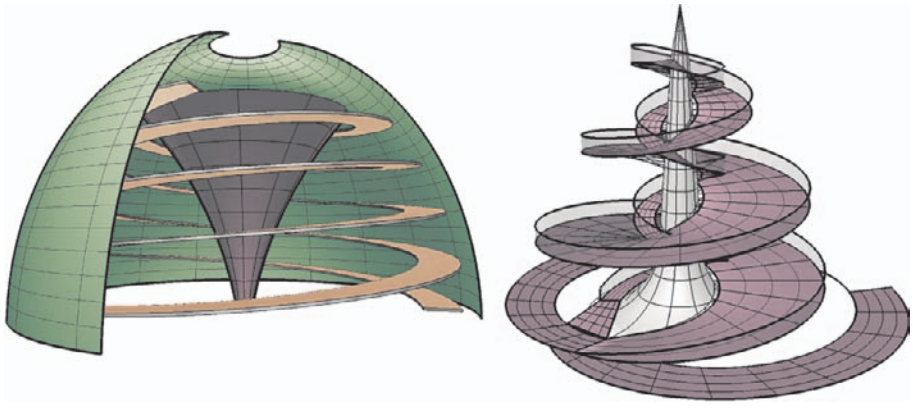

FIGURE 3.37. The output of "reichstag.cpp" and "marielas_building.cpp".

variable Data of type *Cad3Data* and write the following in the Init( ) part of our program:

Listing from "candlestick.cpp":

```
Data.ReadNewMember( "DATA/LLZ/candlestick1.llz" );
Data.ReadNewMember( "DATA/LLZ/candlestick2.llz" );
Data.SetAllColors( Brown );
Data[0].Translate( 0, 0, -H2 - 3.5 );
Data[1].Translate( 0, 0,  H2 + 4.8 );
```

The translation of the objects takes into account the total height of the candlestick and the height of the CAD3D objects we imported. The shading will be done by the command Data.Shade( ) in the Draw( ) part.

The next surface parts are two surfaces of revolution. They are symmetric with respect to the horizontal plane σ through the midpoint of the object (we used $\sigma = [x, y]$). In Init( ), we define only *one* of them as described in [14], page 167. In Draw( ) we write these lines:

Listing from "candlestick.cpp":

```
MiddlePart.Shade( SMOOTH, REFLECTING );
MiddlePart.Reflect( XYplane );
MiddlePart.Shade( SMOOTH, REFLECTING );
```

FIGURE 3.38. A photo of the original candlestick (left) and the output of `"candlestick.cpp"` (right).

and see both of surfaces.[13]

The most interesting object part is probably the center of the object. It consists of four elegantly curved lines connecting the two surfaces of revolution. In a first step we have to find a parameterized equation for one of them (we will call it c). It is of a shape that suggests a parameterization in cylindrical coordinates. By considerations very similar to those made in Example 3.19, we obtain

$$\varrho(u) = r_{\max}\cos u + 4u^2\frac{(r_{\min}-\varepsilon)}{\pi^2}, \quad z(u) = pu^3 + u\frac{4H - p\pi^3}{4}.$$

The real constants $r_{\max}$, $r_{\min}$, p, and H determine the shape of the curve (maximal and minimal radius, slope and total height) while ε is just a small real number to correct the minimal radius. Given the parametric representation of c we can easily implement the corresponding tubular surface HelicalTub. Finally, we rotate and shade it four times in Draw( ):

Listing from `"candlestick.cpp"`:

```
for ( i = 0; i < N; i++ )
{
   HelicalTub.Rotate( Zaxis, i * angle );
```

[13]Note that this in possible only because the reflection ϱ on $[x, y]$ is idempotent, i.e., $\varrho \circ \varrho = \mathrm{id}$. Otherwise, the object transformation in Draw( ) would yield unwanted results.

```
    HelicalTub.Shade( SMOOTH, REFLECTING );
}
```

♢

Example 3.21. Spiral stove
For the modeling of our next surface we are face with a special task: We want to connect two arbitrary parameterized curves so that the curvature changes steadily. That is, the second derivative has no singularities (C^2-continuity). We have already done something similar with Bézier curves in Example 2.34. Here, too we will use a Bézier curve as connecting curve.

The radius of curvature of a Bézier curve at the start or end point is given by

$$\varrho = \frac{\mathsf{n}}{\mathsf{n}-1}\frac{\mathsf{a}^2}{\mathsf{h}}.$$

The meaning of the parameters a and h is explained in Figure 3.39. Note that the control point B_2 may still vary on a straight line parallel to B_0B_1 (i.e., the length b may be changed) without affecting the curve's radius of curvature ϱ in B_0. The successive control points $B_3, B_4, \ldots$ have no impact on ϱ.

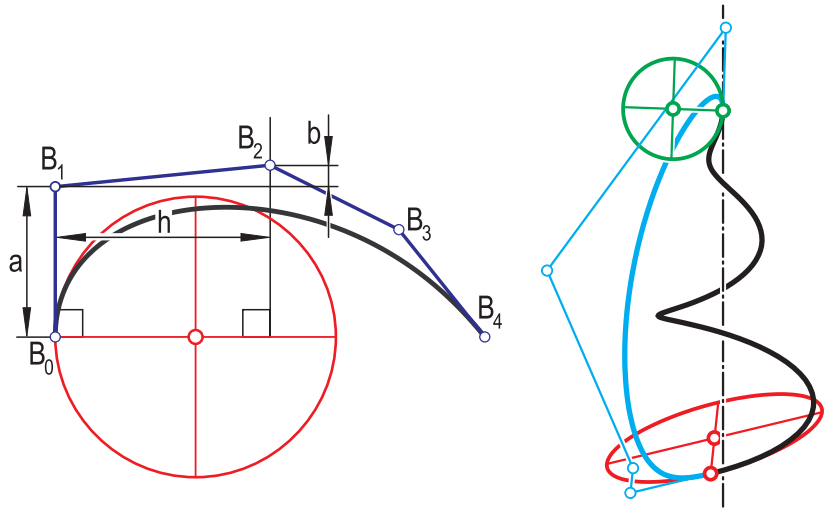

FIGURE 3.39. The osculating circle at the start point of a Bézier curve and a C^2-connection of start and end point of a helispiral.

In `"osculating_curves.cpp"` we implement a convenient method for constructing a C^2-continuous Bézier curve. We derive a class from *ParamCurve3d* in

the usual way (in our example a helispiral curve). In addition, we implement a method to set the three starting points of the Bézier curve.

```
class MyCurve: public ParamCurve3d
{
public:
  P3d CurvePoint( Real t )
  {
    const Real r = 3, c = 0.1;
    Real x, y, z;
    x = r * c * t * cos( t );
    y = r * c * t * sin( t );
    z = 10 * c * t;
    return P3d( x, y, z + 7 );
  }
  void C2Continuation( Real t, Real a, Real b,
                       P3d &A, P3d &B, P3d &C, int n )
  {
    A = CurvePoint( t );
    B = Tangent( t ).InBetweenPoint( a );
    Circ3d osc_circle;
    GetOsculatingCircle( t, NoColor, 1, osc_circle );
    Real h = n / ( n - 1.0 ) * a * a / osc_circle.GetRadius( );
    Plane p( A, osc_circle.axis );
    V3d v1 = B - A, v2 = osc_circle.Middle - A;
    v1.Normalize( );
    v2.Normalize( );
    C = B + h * v2 + b * v1;
  }
};
```

The input parameters of the function C2Continuation(...) are the parameter value t of the curve, the design parameters a and b (compare Figure 3.39), the first three control points A, B, and C and the order n of the Bézier curve. We get the necessary curvature information from the curve's osculating circle, compute the real number h and perform the construction of Figure 3.39 in the osculating plane of the curve point in question.

We draw the helispiral in the interval $[u0, 0]$ and construct the continuation Bézier curve in Draw( ):

```
P3d P[6];
Curve.C2Continuation( U0, -3,  0, P[0], P[1], P[2], 5 );
Curve.C2Continuation( 0,  3, -9, P[5], P[4], P[3], 5 );
BezierCurve.Def( Blue, 200, 6, P );
```

Actually, we connect the start and end points of the helispiral. The construction itself is very simple, but we have to pay attention to the correct order of the control points and the correct sign of the second input parameter a. We must

have $\mathsf{a} < 0$ is necessary for a continuation at the start point, while $\mathsf{a} > 0$ is needed at the end point. Finally, we draw the common osculating circles and some additional lines to get a nice illustration (Figure 3.39).

Now we return to the actual modeling of a real-world surface: a specially designed stove (`"spiral_stove.cpp"`). It has the shape of a slightly deformed ball that turns into a (more or less) tubular surface with a helispiral spine curve. The helispiral shape is chosen because it permits a free flow of smoke. It is necessary to connect the stove to the chimney by means of normed tubes. Thus, the helispiral tube must finally change to a cylinder of revolution with vertical axis and given radius r_1.

For the actual Open Geometry model we mainly use a channel surface. Only the bottom part will be a hemissphere of radius r_0. The spine curve of the surface consists of three parts: Two Bézier curves b_1 and b_2 and a helispiral curve h (Figure 3.41). The tangent of b_1 at the start point and of b_2 at the end point is the z-axis. This allows the connection to the normed tube and to the bottom hemissphere. Furthermore, the Bézier curves are C^2-continuous transition curves to h (as in the previous example) in order to guarantee a smooth channel surface.

The radius function $r(t)$ should ideally have the shape displayed in Figure 3.40 ($[\tau_0, \tau_1]$ is the parameter interval we use for the midcurve). It starts with the initial radius r_0 and quickly approaches the final radius r_1. The shape of the curve suggests the function

$$r(t) = (r_0 - r_1)e^{-c(x-\tau_0)} + r_1$$

with $c \in \mathbb{R}^+$.

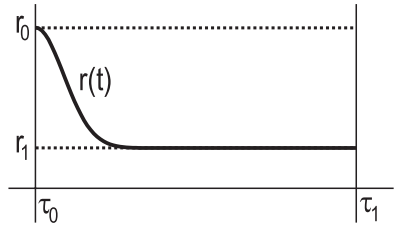

FIGURE 3.40. The radius function.

At the very beginning of our program we decide on the initial and final radii R_0 and R_1 of the channel surface and on the parameter interval $[\mathsf{T0}, \mathsf{T1}]$ of the helispiral midcurve. The CurvePoint(...) function of the helispiral reads as follows:

Listing from `"spiral_stove.cpp"`:

P3d CurvePoint(*Real* t)

```
{
    const Real r = 3, c = 0.2;
    Real x, y, z;
    x = r * c * t * cos( t );
    y = r * c * t * sin( t );
    z = 10 * c * t;
    return P3d( x, y, z );
}
```

Of course, we implement the method for constructing C^2-continuous Bézier curves of `"osculating_curves.cpp"` as well. Our total t-parameter interval for the channel surface will be $[\mathsf{T0}-1, \mathsf{T1}+1]$ ($[\mathsf{T0}, \mathsf{T1}]$ for the helispiral and two times an interval of length 1 for the Bézier curves). Therefore, the radius and midpoint functions for the spiral surface are as follows:

Listing from `"spiral_stove.cpp"`:

```
Real rad_function( Real v )
{
    v = v - T0 + 1;
    return ( R0 - R1 ) * exp( -5 * v * v ) + R1;
}
P3d mid_function( Real v )
{
    if ( v <= T0 )
    {
        return BezierCurve1.CurvePoint( v + 1 - T0);
    }
    if ( T0 < v && v < T1 )
        return MidCurve.CurvePoint( v );
    else
        return BezierCurve2.CurvePoint( v - T1 );
}
```

The Bézier curves must, of course, be defined in Init( ). The z-axis is tangent to both curves:

Listing from "spiral_stove.cpp":

```
P3d P1[5], P2[5];
MidCurve.C2Continuation( T0, -3, 0, P1[4], P1[3], P1[2], 4 );
P1[1].Def( 0, 0, -15 );
P1[0].Def( 0, 0, -22 );
BezierCurve1.Def( Blue, 200, 5, P1 );

MidCurve.C2Continuation( T1, 3, 0, P2[0], P2[1], P2[2], 4 );
P2[3].Def( 0, 0, 5 );
P2[4].Def( 0, 0, 10 );
BezierCurve2.Def( Green, 200, 5, P2 );
```

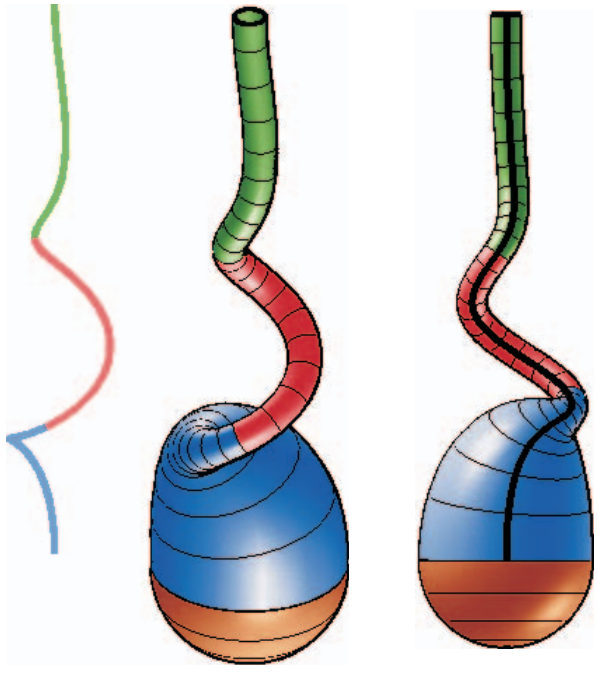

FIGURE 3.41. The spiral stove.

We define three parts of the channel surface in order to be able to use different colors. Furthermore, we translate the three midcurve parts a little and draw them in the corresponding colors. The output of the program is displayed in Figure 3.41. ♢

3.5 Spline Surfaces

Open Geometry 2.0 not only supports Bézier curves and B-spline curves (see Section 2.6) but Bézier surfaces and B-spline surfaces as well. Curves and surfaces are defined and implemented in a more or less analogous way. However, there is an important difference: *The spline surfaces are defined via a 2D array of control points.*

This makes dynamic memory allocation much harder and has an effect on the use of the corresponding classes in an Open Geometry program. Fortunately, Open Geometry can support you with that task. In the following we will shortly describe the main properties of Bézier surfaces and spline surfaces and present a few examples. Since most properties are analogous to the corresponding 2D classes, we do not go too much into detail.

Bézier surfaces

Integral Bézier surfaces are defined by $(m+1)(n+1)$ control points B_{ij}; for rational Bézier surfaces we can additionally prescribe an array ω_{ij} of weights $(i = 0, \ldots, m,\ j = 0, \ldots, n)$. The following (very basic) properties are important for the use of Bézier surfaces for design purposes (compare Figure 3.42):

- The border lines are Bézier curves with control polygons $B_{00}, \ldots, B_{m0}$, $B_{m0}, \ldots, B_{mn}$, $B_{mn}, \ldots, B_{0n}$ and $B_{0n}, \ldots, B_{00}$, respectively.
- The surface contains the points B_{00}, B_{m0}, B_{0n}, and B_{mn} of the control net.
- The tangent plane in B_{00} is spanned by B_{00}, B_{01}, and B_{10}. Analogously we can determine the tangent planes in B_{m0}, B_{0n}, and B_{mn}.
- The control net gives a rough idea of the shape of the Bézier surface.
- The μ-th partial derivatives of the surface along the border line defined by $B_{00}, \ldots, B_{m0}$ depend on the points B_{ij} with $0 \le j \le \mu$ only.[14]

For the computation of a surface point $X = X(u, v)$, Open Geometry uses a generalization of DeCasteljau's algorithm to two-dimensional arrays. That is, we apply the algorithm to the u-threads of the control nets and produce a control polygon $B_0^*, \ldots, B_n^*$. Then we apply the original algorithm to this polygon and get the corresponding surface point. Of course, it is also possible to start with the v-threads of the control net.

The Open Geometry classes *BezierSurface* (and *RatBezierSurface*) provide more or less the same methods as the corresponding Bézier curves. That is, we can split them, elevate the degree, separately change a control point or

[14] This is important for the construction of smooth blending surfaces.

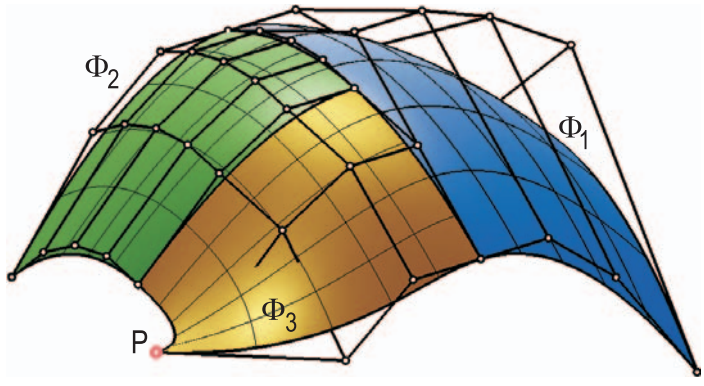

FIGURE 3.42. We split a Bézier surface into three patches Φ_1, Φ_2, and Φ_3. Then we elevate the degree of Φ_2 and change the control point P.

weight, draw the control net, and mark the control points. Occasionally, we have to specify a parameter direction (SplitU(...) and SplitV(...)). You can learn their usa by having a look at the two sample files `"bezier_surface.cpp"` and `"rat_bezier_surface.cpp"`. A few sample code lines can be found on pages 484 and 522.

In this place it is necessary to draw your attention to a very important point: *The number of control points is not known before the definition of the surface.* Therefore, it is necessary to allocate the memory for them dynamically. As long as only one-dimensional arrays are needed (e.g., for Bézier curves), the user need not be aware of this. C++ syntax does not differ from the standard case. Here, however, you have to use *arrays of pointers to control points.* We display the important code lines of `"bezier_surface.cpp"`:

Listing from `"bezier_surface.cpp"`:

```
BezierSurface BezSurf[3];

void Scene::Init( )
{
    const int m = 4, n = 3;
    P3d **P = NULL;
    ALLOC_2D_ARRAY( P3d, P, m, n, "P" );

    ...
    // define the Bézier surfaces
    BezSurf[0].Def( Blue, 31, 61, m, n, P );
    ...
```

```
    FREE_2D_ARRAY( P3d, P, 4, 3, "P" );
}
```

We define an array of pointers to a *P3d* object and we allocate and free the memory in Init( ) by using OPEN GEOMETRY macros. This method is both *simple* and *safe*. You need not bother about the complex task of allocating the memory yourself. If you want to use the control points globally, you have to move the code line that frees the memory to CleanUp( ). This is one of the rare occasions where this part of an OPEN GEOMETRY program is really necessary.[15]

Displaying rational Bézier surfaces is almost as easy. The only difference is that you have to reserve additional memory for a 2D array of reals (the weights). The corresponding listing from "rat_bezier_surface.cpp" reads as follows:

Listing from "rat_bezier_surf.cpp":

```
RatBezierSurface BezSurf[3];

void Scene::Init( )
{
    const int m = 4, n = 3;
    P3d **P = NULL;
    Real **weight = NULL;
    ALLOC_2D_ARRAY( P3d, P, m, n, "P" );
    ALLOC_2D_ARRAY( Real, weight, m, n, "weight" );

    ...
    // define the Bézier surfaces
    BezSurf[0].Def( Blue, 31, 61, m, n, P, weight );
    ...

    FREE_2D_ARRAY( P3d, P, m, n, "P" );
    FREE_2D_ARRAY( Real, weight, m, n, "weight" );
}
```

[15] If you use dynamic memory allocation, you will usually write a class of your own with appropriate destructor. That is, the freeing process is handled by class methods and is not directly visible.

Example 3.22. Cube corner
Even the simple task of designing a more or less cube-shaped object can be challenging. For a student's project we had to display a cube with rounded edges. The transition between the rounding surface and the cube's faces had to be very smooth. The designer wanted to avoid any visible discontinuities. Therefore, we could not simply use quarters of cylinders of revolution: A shadow falling on the rounded edge would have had a break point.

In order to avoid this, we need at least C^2-continuity. Therefore, we decided to use cylinders with a Bézier curve as base line. The order of the base line b must be at least five. Figure 3.43 shows a possible configuration of the control points P_i.

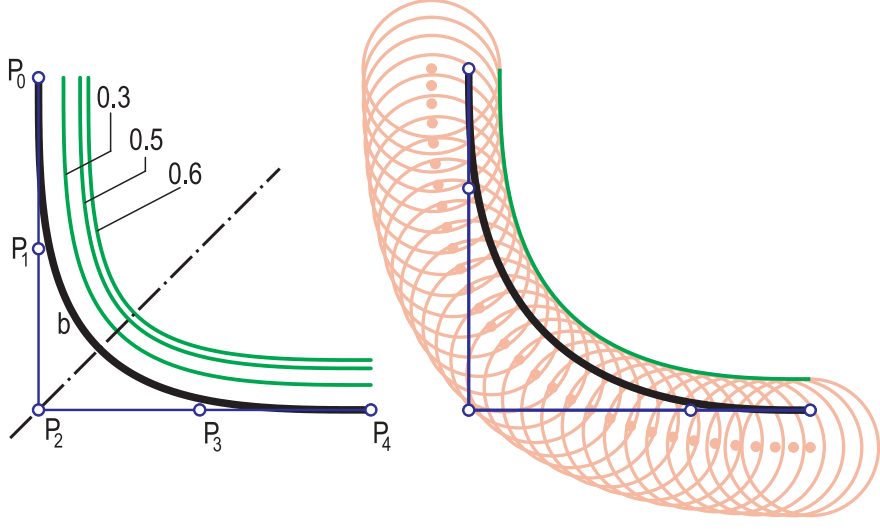

FIGURE 3.43. The profile of the rounding cylinder with control points and parallel curves.

The triples $\{P_0, P_1, P_2\}$ and $\{P_2, P_3, P_4\}$, respectively, are collinear. As a design parameter we can choose P_1 somewhere on the straight line through P_0 and P_2. The corresponding point P_3 is determined by the configuration's symmetry.

With `"cube_corner1.cpp"` we want to solve the following tasks:

- Visualize the cube corner with the rounded edges and all intersection curves.
- Allow interactive change of the design parameter to get an attractive and useful shape.
- Return the design parameter of the optimal shape for further testing and use.

We start by defining three congruent Bézier curves XCurve, YCurve, and ZCurve in planes perpendicular to the coordinate axes:

Listing from `"cube_corner1.cpp"`:

```
P3d P[N];
P[0].Def( -L, -A, 0 );
P[1].Def( -L, -f * A, 0 );
P[2].Def( -L, 0, 0 );
P[3].Def( -L, 0, -f * A );
P[4].Def( -L, 0, -A );
XCurve.Def( Black, 200, N, P );
```

In the above listing, L and A are global parameters to determine the length of the rounding cylinders and the distance of P_0 and P_4 from the cube's edge. The integer N gives the order of the Bézier curve, and f is a local parameter that is responsible for the positions of P_1 and P_3. The control points of YCurve and ZCurve are determined by transforming the control points of XCurve:

Listing from `"cube_corner1.cpp"`:

```
int i;
for ( i = 0; i < N; i++ )
{
   P[i].Rotate( Zaxis, 90 );
   P[i].Reflect( YZplane );
}
YCurve.Def( Black, 200, N, P );

for ( i = 0; i < N; i++ )
{
   P[i].Rotate( Xaxis, 90 );
   P[i].Rotate( Zaxis, 90 );
}
ZCurve.Def( Black, 200, N, P );
```

Now we have to find the intersection curves of the rounding cylinders. Again, they lie on plane Bézier curves. We find their control points by projecting the control points of XCurve, YCurve, and ZCurve in planes of symmetry of the coordinate axes:

Listing from "cube_corner1.cpp":

```
StrL3d s;
Plane a;
a.Def( Origin, V3d( 1, -1, 0 ) );
P3d XY[N];
for ( i = 0; i < N; i++ )
{
    s.Def( XCurve.GetControlPoint( i ), Xdir );
    XY[i] = a * s;
}
XYCurve.Def( Black, 50, N, XY );
```

The same is done for the two remaining intersection curves. The cylinders themselves are parameterized surfaces. We define them, already taking care of the intersection curves:

Listing from "cube_corner1.cpp":

```
class MyXYCylinder: public ParamSurface
{
public:
    virtual P3d SurfacePoint( Real u, Real v )
    {
        if ( u <= 0.5 )
            return ( 1 - v ) * XCurve.CurvePoint( u ) +
                        v * XYCurve.CurvePoint( u );
        else
            return ( 1 - v ) * XCurve.CurvePoint( u ) +
                        v * ZXCurve.CurvePoint( u );
    }
};
MyXYCylinder XYCyl;
```

So far, we have done everything necessary for the visualization. In order to be able to change the surface shape interactively, we put the defining methods of all Bézier curves and the rounding cylinders into a function

void ComputeSurfaces(*Real* f),

where f is the value of the design parameter. For the animation we use two global variables: a real F and a variable of type *Boolean* called ComputeAgain. Init( ) and Animate( ) now read

Listing from "cube_corner1.cpp":

```
void Scene::Init( )
{
   F = 0.0;
   ComputeAgain = false;
   ComputeSurfaces( F );
}
```

and

Listing from "cube_corner1.cpp":

```
void Scene::Animate( )
{
   if ( ComputeAgain )
   {
      ComputeSurfaces( F );
      ComputeAgain = false;
   }
}
```

The values of F and CompteAgain can change only in a small part written in Draw( ):

Listing from "cube_corner1.cpp":

```
int key = TheKeyPressed();
switch (key )
{
case 'f':
   F += 0.02;
   ComputeAgain = true;
   break;
case 'j':
   F -= 0.02;
   ComputeAgain = true;
   break;
}
```

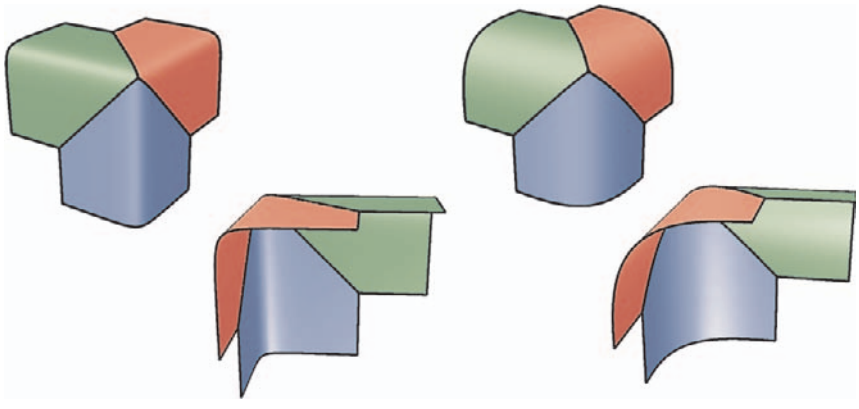

FIGURE 3.44. Differently shaped cube corners.

That is, if you start the animation and press the key **f** or **j** you can change the appearance of the cube corner. Finally, we print the current value of the design parameter on the screen in order to fulfill the last task of the program. Two possible shapes of the cube corner are displayed in Figure 3.44.

After having decided on the optimal shape with respect to technical and optical criteria, the designer encounters other problems. He has to produce a model of the object, and again, OPEN GEOMETRY can help considerably. In order to produce a metal template, the curve's profile plus some data for the milling tool are needed. We write the necessary routines in `"cube_corner2.cpp"`. At the top of the file we find the global constants that determine the length between the cube's edge and the beginning of the profile curve, the thickness of the material, the radius of the milling tool, and the number of discrete positions of the tool that have to be computed.

Listing from `"cube_corner2.cpp"`:

```
const Real Length = 5.5;
const Real Distance = 0.5;
const Real MillingRad = 1.1;
const int N = 31;
```

In Init( ) we define the profile curve according to the consideration of the `"cube_corner1.cpp"` example. Using the method GetOffset(...) of *ParamCurve2d*, we can easily display the offset curve at the given distance, taking into account the thickness of the material. It is necessary to check that it is not curved too much and has no singularities.

Listing from `"cube_corner2.cpp"`:

```
const Real a = 0.65 * Length;
const int n = 5;
P2d P[n];
P[0].Def( 0, Length );
P[1].Def( 0, a );
P[2].Def( 0, 0 );
P[3].Def( a, 0 );
P[4].Def( Length, 0 );
BezierCurve.Def( Black, 200, n, P );
OffsetCurve.Def( Green, 200 );
BezierCurve.GetOffset( Distance, 0, 1, OffsetCurve );
```

Besides displaying the relevant curves, we perform another task in Draw( ): We compute different positions of the milling tool and mark them with small circles (Figure 3.43). The circle centers lie on another offset curve of the profile. The milling operator has to know their coordinates, so we write them in `"try.log"`. Note that the vertex of the control polygon of the profile curve coincides with the center of our coordinate system ($P_2 = (0,0)^t$). This choice gives the most convenient coordinates for the milling process. ♢

B-spline surfaces

B-spline surfaces and rational B-spline surfaces (NURBS surfaces) are the most frequent surface type in today's CAGD, and Open Geometry 2.0 supports them as well. Please, make sure that you are familiar with the contents of the section on B-spline curves (page 150) before reading on.

In order to define a B-spline surface, you need a 2D array

$$\{\vec{d}_{ij} \mid i = 0, \ldots, n_1, j = 0, \ldots, n_2\}$$

of control points, *two* knot vectors

$$\mathbf{T}_1 = \{0, \ldots, n_i - 1, n_i, n_i + 1, \ldots, n_i + k_i\}, \quad i = 1, 2,$$

and a 2D array of weights ω_{ij} (in the case of a rational surface). In parameterized form, the integral B-spline surface is described by

$$\Phi \ldots \vec{X}(u,v) = \sum_{i=0}^{n_1} \sum_{j=0}^{n_2} \vec{d}_{ij} N_i^{k_1}(u) N_j^{k_2}(v),$$
$$u \in [t_{k_1-1}, t_{n_1+1}], \quad v \in [t_{k_2-1}, t_{n_2+1}],$$

where $N_i^k(u)$ is defined in equation (9) of Chapter 2. However, this is not really important to us because we can compute the surface point $\vec{X}(u_0, v_0)$ in the following way:

1. We apply the algorithm of Cox–De Boor (see page 157) for $u = u_0$ to the u-threads $\{\vec{d}_{ij} \mid i_0 = \text{const.}\}$ of the control net $\{\vec{d}_{ij}\}$. This yields a sequence $\{\vec{e}_j\}$ of auxiliary control points.
2. We apply the Cox–De Boor algorithm for $v = v_0$ to the sequence $\vec{e}_j$ and gain the surface point $\vec{X}(u_0, v_0)$.

Rational B-spline surfaces can be obtained by applying these algorithms in 4D and by projecting the resulting surfaces in 3-space.

We now give an example of the use of the Open Geometry class *NUBS_Surf*. The class *NURBS_Surf* is employed in a completely analogous way. The slightly different defining methods are listed in Chapter 6, page 501.

Example 3.23. NUBS surface
In Chapter 2, page 159, we described how to generate open and closed B-spline curves. When we deal with B-spline surfaces we can close the u- and/or v-threads of the control nets and get four types of surfaces (open–open, closed–open, open–closed, closed–closed). In `"nubs_surf.cpp"` we will generate a random sequence of NUBS surfaces of all four types.[16] We use the two global variables

```
NUBS_Surf Surf;
RandNum Rnd( -1, 1 );
```

The rest of the program is contained in Draw( ). This means that a new surface will be computed with every new frame. At first, we must reserve the memory for the control points. We use the Open Geometry macro ALLOC_2D_ARRAY for that task:

Listing from `"nubs_surf.cpp"`:

```
int pnum1 = 5, pnum2 = 7; // dimension of control nets

// allocate memory for the control points
P3d **P = NULL;
ALLOC_2D_ARRAY( P3d, P, pnum1, pnum2, "P" );
```

Now we define the B-spline surface. Initially, the control points are located on a torus before they are translated in a random direction. The values of the two Boolean variables close_u and close_v are also determined at random.

[16] `"nurbs_surf.cpp"` does the same for a rational B-spline surface.

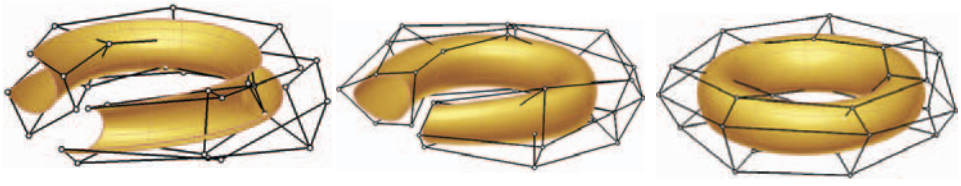

FIGURE 3.45. NUBS surfaces that are closed in zero, one, and two parameter directions, respectively.

Listing from `"nubs_surf.cpp"`:

```
int i, j;
for ( i = 0; i < pnum1; i++ )
   for ( j = 0; j < pnum2; j++ )
   {
      P[i][j].Def( 3, 0, 0 );
      P[i][j].Rotate( Yaxis, i * 360.0 / pnum1 );
      P[i][j].Translate( 10, 0, 0 );
      P[i][j].Rotate( Zaxis, j * 360.0 / pnum2 );
      P[i][j].Translate( Rnd( ), Rnd( ), Rnd( ) );
   }
Boolean close_u = Rnd( ) > 0;
Boolean close_v = Rnd( ) > 0;
Surf.Def( Yellow, 50, 50, pnum1, pnum2, P, 3, 3, close_u, close_v );
Surf.PrepareContour( );
```

The two integers after the input parameter P determine the continuity class of the u- and v-parameter lines, respectively. In our case, the surface will be of class C^3. Now we immediately free the memory by writing

```
FREE_2D_ARRAY( P3d, P, pnum1, pnum2, "P" );
```

Finally, we shade the surface and draw some parameter lines and the control net.

Listing from "nubs_surf.cpp":

```
Surf.Shade( SMOOTH, REFLECTING );
Surf.MarkControlPoints( Black, 0.25, 0.15 );
Surf.DrawControlPolygon( Black, MEDIUM );
Surf.Contour( Brown, MEDIUM );
Surf.WireFrame( Brown, 10, 10, THIN );
if ( !close_v )
    Surf.ULines( Brown, 2, MEDIUM );
if ( !close_u )
    Surf.VLines( Brown, 2, MEDIUM );
```

In Figure 3.45 you can see the results for different values of close_u and close_v. ♢

3.6 Simple 3D Animations

One of the best things in Open Geometry is the possibility of creating animated scenes. Open Geometry offers a host of tools for transforming the objects of your scenes. Therefore, animations of moderate complexity can easily be done. This chapter will present a few examples of that kind. More advanced animations are left to Sections 4.1 and 4.5.

Example 3.24. Planetary paths
The apparent path curves of planets against the firmament are of a strange shape. Their weird loops amazed professional astronomers a few hundred years ago and hobby astronomers of today. The *Deutsches Museum München* exhibits a mechanical apparatus to explain their generation (Figure 3.46).

Of course, we have to write an Open Geometry animation of that. The corresponding program is "planet_path.cpp". There, we visualize the path of Mercury on an imaginary sky sphere. We chose Mercury because the supporting plane of its path circle[17] intersects the supporting plane of Earth's path circle in the (rather big) angle of about 7°. As a consequence, the resulting path curve will be more impressive.

We start the program by defining three spheres: Sun, Earth, and Mercury. The initial position of Mercury is chosen with respect to real astronomical dimension.

[17]Of course, Mercury's path is not a circle but an ellipse. For reasons of simplicity, we will approximate it by a circle.

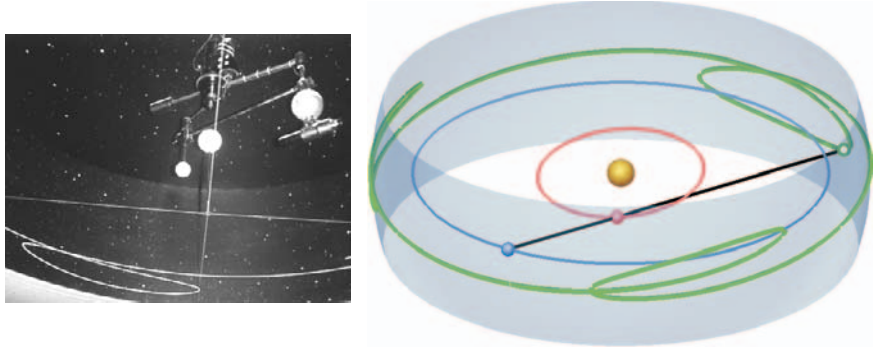

FIGURE 3.46. A mechanical device to display the apparent path curve of a planet in the *Deutsches Museum München* (left) and its OPEN GEOMETRY simulation.

Listing from `"planet_path.cpp"`:

```
Sun.Def( Yellow, Origin, 1, 10, 10 );
Earth.Def( Blue, P3d( 15, 0, 0 ), 0.5, 8, 8 );

Mercury.Def( Red, P3d( 6, 0, 0 ), 0.5, 8, 8 );
Real phi = 7;
Mercury.Rotate( Yaxis, phi );
MercuryAxis = Zaxis;
MercuryAxis.Rotate( Yaxis, phi );
Mercury.Rotate( MercuryAxis, 30 );
```

In Animate( ) we will rotate Earth and Mercury about their respective axes. The ratio of angular velocities should actually be $88 : 365$. We modified it to $91 : 364$ in order to achieve a closed path curve after one terrestrial year. We display only the equatorial zone of the sky sphere because we do not want to disturb the view of what is happening. Therefore, we use a parameterized surface with the parameter interval $(u, v) \in [-t\pi, t\pi] \times [360]$ and $t = 0.1$. Later, in Draw( ), we will shade it by using some transparency.

Listing from `"planet_path.cpp"`:

```
class MySkySphere: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real u, Real v )
   {
```

```
        P3d P( 0, Radius * cos( u ), Radius * sin( u ) );
        P.Rotate( Zaxis, v );
        return P;
    }
};
MySkySphere SkySphere;
```

The most interesting part of Draw( ) is this:

Listing from `"planet_path.cpp"`:

```
StrL3d proj_ray;
proj_ray.Def( Earth.GetCenter( ), Mercury.GetCenter( ) );

Sphere sky_sphere;
sky_sphere.Def( NoColor, Origin, Radius );
int n;
P3d A, B;
n = sky_sphere.SectionWithStraightLine( proj_ray, A, B );
if( proj_ray.GetParameter( A ) < 0 )
    A = B;
StraightLine3d( Black, Earth.GetCenter( ), A, MEDIUM );
A.Mark( Green, 0.2, 0.1 );
MercuryPath.AddPoint( A );
MercuryPath.Draw( THICK );
```

We define the projection ray through the centers of Earth and Mercury and intersect it with the sky sphere. In order to be able to use the SectionWithStraightLine(...) method of *Sphere*, we define the sky sphere as a *Sphere* object as well. We ensure that the correct intersection point (the point with positive parameter value on the projection ray) is always stored in A and add this point to the apparent path curve. The resulting image shows the typical loops of the planetary path. ♢

Example 3.25. Circle caustics of higher order
In Example 2.32 we presented the n-th caustic c_n of a conic with respect to a general light source E. Here we will demonstrate how to generate the n-th caustic of a circle with the help of a reflecting cylinder ζ. Depending on the position of the light source E and the height h of ζ, a light ray will be reflected n times before it intersects the base plane of ζ. Thus, the n-th circle caustic will be visible.

In reality, at most one-half of the caustic occurs. Usually it is even less than one-half, because the light rays have to enter through the top circle. For our program (`"reflecting_cylinder.cpp"`) we will make two assumptions:

- The cylinder ζ is a one-way mirror, i.e., light rays can enter the interior but are then reflected inside ζ.
- The light source E is a point at infinity.

The second assumption yields two nice results (compare Figure 3.47)

1. The n-th caustic c_n is a trochoid (compare [14], page 239). Let r be the radius of ζ and a and b the radii of the fixed and rolling circles, respectively. Then we have
$$a + 2b = r \quad \text{and} \quad \frac{a}{b} = \frac{2}{2n-1}. \tag{6}$$
2. The primary image c_0 of c_n is an ellipse.[18]

Now we are going to visualize these results. We start with the caustic c_n:

Listing from `"reflecting_cylinder.cpp"`:

```
class MyCaustic: public ParamCurve3d
{
public:
  P3d CurvePoint( Real u )
  {
    Real a = Radius / ( 2 * ReflNum );
    Real b = ( Radius - a ) / 2;
    Real t = ( a + b ) / b;
    P3d P( ( a + b ) * cos( u ) + b * cos( t * u ),
           ( a + b ) * sin( u ) + b * sin( t * u ), 0 );
    return P;
```

[18]These results are due to a public talk given by C. Engelhart from Munich's Technical University. The primary image c_0 is defined as the set of all points $X \in \zeta$ such that EX intersects c_n after n reflections in ζ.

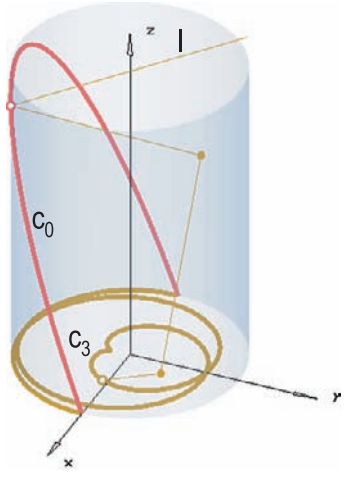

FIGURE 3.47. A ray of light l is reflected three times at a cylinder of revolution and intersects the catacaustic c_3. Its primary image c_0 on the cylinder is an ellipse.

```
    }
    void Def( Color c, int n, Real umin, Real umax, int N )
    {
        ReflNum = N;
        ParamCurve3d::Def( c, n, umin, umax );
    }
private:
    int ReflNum;
};
MyCaustic Caustic;
```

The number n of reflections is stored in the private integer variable ReflNum and has to be passed together with the definition of the caustic. For the parametric representation we solved the formulas (6) for a and followed considerations in [14].

We use a global variable N to store the number of reflections. During the animation we will be able to change this value interactively. Therefore, we have to define the caustic in Draw( ):

Listing from `"reflecting_cylinder.cpp"`:

```
Caustic.Def( Yellow, N * 100, Umin, Umax, N );
Caustic.Draw( MEDIUM );
```

We draw only one-half of the caustic (the part that can actually occur). For this we have to take umin $= (1 - 2\mathsf{N})\pi$ and umax $= 0$. The higher the number N of reflections, the longer the caustic will be. Hence the total number of computed curve points depends on N as well.

Now we compute the primary image of c_n on ζ. For this task we will use a special function:

Listing from `"reflecting_cylinder.cpp"`:

```
void ReflectAtCylinder( StrL3d &s )
{
   V3d d = s.GetDir( );
   P3d P = s.GetPoint( );
   int n;
   Real t1, t2;
   n = QuadrEquation( d.x * d.x + d.y * d.y, 2 * P.x * d.x + 2 * P.y * d.y,
                      P.x * P.x + P.y * P.y - Radius * Radius, t1, t2 );
   if ( n )
   {
      P3d S = s.InBetweenPoint( max( t1, t2 ) );
      d.Reflect( Plane( S, V3d( S.x, S.y, 0 ) ) );
      s.Def( S, d );
   }
}
```

The input parameter is a straight line s. We insert the parametric representation of s in the equation $x^2 + y^2 = r^2$ of ζ and compute the parameter values of the intersection points by solving the resulting quadratic equation. Then we use the fact that in Open Geometry every *StrL3d* object is equipped with an orientation: Since we deal only with reflection on the inside of ζ, we can always take the intersection point S that corresponds to the bigger parameter value. Finally, we reflect s at the tangent plane of ζ in S and redefine s. Note that the "new" line s will have the right orientation! Now we can implement the image c_0 of c_n on the reflecting cylinder:

Listing from `"reflecting_cylinder.cpp"`:

```
class MyCausticImage: public ParamCurve3d
{
public:
   P3d CurvePoint( Real u )
   {
      P3d C = Caustic.CurvePoint( u );
      V3d dir = Caustic.TangentVector( u );
```

```
        if ( u < critical_u )
            dir *= -1;
        dir.z = DeltaZ;
        StrL3d s( C, dir );
        int i;
        for ( i = 0; i < ReflNum; i++ )
            ReflectAtCylinder( s );
        return s.GetPoint( );
    }
    Real CriticalU( void )
    {
        return critical_u;
    }
    void Def( Color col, int m, Real umin, Real umax, int n )
    {
        ReflNum = n;
        critical_u = ( 0.5 - ReflNum ) * PI;
        ParamCurve3d::Def( col, m, umin, umax );
    }
private:
    int ReflNum;
    Real critical_u;
};
MyCausticImage CausticImage;
```

Here we have two private member variables: the total number of reflections and a critical parameter value that depends on this number. The main idea of the CurvePoint(...) function tracing back the path of the incoming ray of light. We start with the caustic point and the corresponding tangent vector dir. The z-value of dir is set to a value DeltaZ that remains constant during the program. It determines the position of the light source E in $[y, z]$. DeltaZ and the incidence angle α of the light rays with $[x, y]$ satisfy the relation DeltaZ $= \tan\alpha$.

We have to take care to use the correct orientation of the light ray at the beginning of the backward ray-tracing. Later, our function ReflectAtCylinder(...) will do this for us. For this purpose we can use the critical parameter value critical_u. It belongs to the cusp of c_n. There, we have to correct the direction of the tangent. We reflect s at the cylinder's surface and finally return the point used for the last redefinition of s in ReflectAtCylinder(...). Again, we have to put the definition of the curve in Draw( ):

Listing from `"reflecting_cylinder.cpp"`:

```
CausticImage.Def( Red, N * 100, Umin, Umax, N );
CausticImage.Draw( MEDIUM );
```

In order to demonstrate the generation of both curves c_n and c_0, we draw the path of one ray of light on the screen. The considerations behind the following lines are identical to those of the CurvePoint(...) function of the primary image c_0:

Listing from "reflecting_cylinder.cpp":

```
V3d dir = Caustic.TangentVector( U );
if ( U < CausticImage.CriticalU( ) )
    dir *= -1;
dir.z = DeltaZ;
P3d C_old, C = Caustic.CurvePoint( U );
C.Mark( Yellow, 0.2, 0.1 );
StrL3d s( C, dir );
int i;
for ( i = 0; i < N; i++ )
{
    ReflectAtCylinder( s );
    C_old = C;
    if ( i > 0 )
        C_old.Mark( Yellow, 0.15 );
    C = s.GetPoint( );
    StraightLine3d( Yellow, C_old, C, THIN );
}
C.Mark( Red, 0.2, 0.1 );
s.Draw( Yellow, 0, 10, THIN );
```

Finally, we define and shade the reflecting cylinder ζ transparently and allow the interactive control of N. The height of ζ will always be enough to provide space for the primary image c_0. The output of "reflecting_cylinder.cpp" is displayed in Figure 3.47.

Listing from "reflecting_cylinder.cpp":

```
ReflCyl.Def( Blue, Radius, Radius, ( 2 * N - 1 ) * Radius * DeltaZ,
    50, HOLLOW, Zaxis );
SetOpacity( 0.2 );
ReflCyl.Shade( SMOOTH, REFLECTING );
SetOpacity( 1 );

// Allow interactive change of reflection number.
int key = TheKeyPressed();
switch (key )
{
case 'f':
    N += 1;
```

```
    Umin = -( 2 * N - 1 ) * PI;
    break;
case 'j':
    if ( N > 1 )
        N -= 1;
    Umin = -( 2 * N - 1 ) * PI;
    break;
}
PrintString( Black, 6, 17.5, "Press <f>or <j>to increase or" );
PrintString( Black, 6, 16.5, "decrease the number of reflections" );
PrintString( Black, 6, 15.0, "number of reflections...%2.0f", N );
```

♢

Example 3.26. Folding conoids

In `"folding_conoids1.cpp"` we visualize a simple mechanical method of generating a pair of right conoids. We start with a bunch s_i of z-parallel lines in $[x, z]$ ($i = 0, \ldots, \mathsf{N}$). Here, $\mathsf{A}[\mathsf{i}]$ and $\mathsf{B}[\mathsf{i}]$ are the intersection points of s_i with two x-parallel straight lines a and b. That is, the points $\mathsf{A}[0]$, $\mathsf{A}[\mathsf{N}]$, $\mathsf{B}[0]$, and $\mathsf{B}[\mathsf{N}]$ are the corners of a rectangle in $[x, z]$.

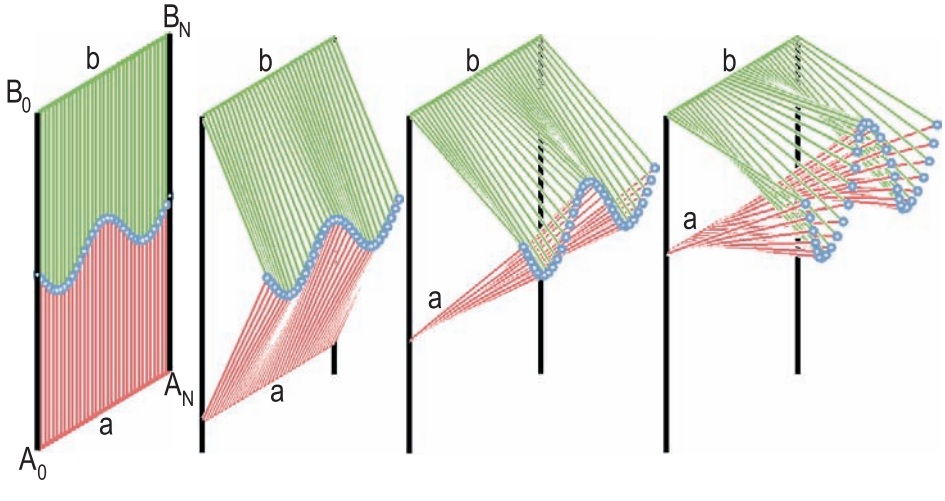

FIGURE 3.48. Folding a pair of right conoids.

Now we choose a joint of revolution $\mathsf{F}[\mathsf{i}]$ on each line s_i. The joint $\mathsf{F}[\mathsf{i}]$ allows a rotation about an axis parallel to x; i.e., we can fold the line segment $\mathsf{A}[\mathsf{i}]\mathsf{B}[\mathsf{i}]$. Now we translate a and the points $\mathsf{A}[\mathsf{i}]$ parallel to z and keep b and the points

B[i] fixed. Then the line segments will fold at the joints F[i] and yield two sets of rulings on right conoids (Figure 3.48).[19]

Reversed, we can (at least locally) generate any right conoid Ψ in this way: Figure 3.49 displays the normal projection of a generator s of Ψ onto a director plane. We consider an arbitrary plane ε parallel to the projection rays. The intersection curve of Ψ and ε is c. Now we choose a curve $c^* \subset \Psi$ such that for any generator s of Ψ the distance between $A := s \cap c$ and $A^* := c \cap c^*$ is of constant length $a+b$ (a and b are arbitrary positive reals); F is the unique point on s with $\overline{AF} = a$ and $\overline{FA^*} = b$. If $b > \overline{F\varepsilon}$, we can find a real point $B \in \varepsilon$ such that $FB = FA^* = b$, and this is still possible in some neighborhood of the generator s. Now the points A, B, and F have the same meaning as A[i], B[i], and F[i] in the preceding considerations.

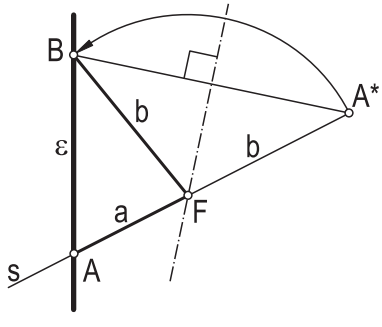

FIGURE 3.49. Any right conoid can be folded.

It is not difficult to imitate this construction mechanically or in an Open Geometry program. The corresponding code is straightforward. Its heart is a function that computes the position of F[i] from the z-coordinate ζ of A[i]:

Listing from `"folding_conoids1.cpp"`:

```
void SetPoints( Real zeta )
{
  int i;
  Real x, y, z;
  for ( i = 0, x = X0; i < N; i++, x += Delta_x )
  {
    z = ( a[i] * a[i] - b[i] * b[i] + ( Z1 + zeta ) * ( Z1 - zeta ) )
      / ( 2 * ( Z1 - zeta ) );
    y = -z * z + 2 * zeta * z + a[i] * a[i] - zeta * zeta;
    if ( y >= 0 )
```

[19] All rulings intersect either a or b and the line at infinity of $[y, z]$.

```
        {
            y = Sqrt( y );
            A[i].Def( x, 0, zeta );
            F[i].Def( x, y, z );
        }
    }
}
```

We have to intersect two circles in a plane parallel to $[y, z]$ centered at A[i] and B[i]. Their radii a[i] and b[i] are determined by the initial position of the joints F[i]. We define them in Init( ):

Listing from "folding_conoids1.cpp":

```
void Scene::Init( )
{
    int i;
    Real x;
    for ( i = 0, x = X0; i < N; i++, x += Delta_x )
    {
        a[i] = cos( x ) + 0.5 * Height;
        b[i] = Height - a[i];
        A[i].Def( x, 0, Z0 );
        B[i].Def( x, 0, Z1 );
        F[i].Def( x, 0, Z0 + a[i] );
    }
}
```

The points X0, X1, Z0, and Z1 are the original x- and z-coordinates of A[0], A[N], B[0], and B[N] (Figure 3.48). Height = Z1 – Z0 is the total height of the rectangle. In order to provide the possibility of an interactive animation we write in Draw( ):

Listing from "folding_conoids1.cpp":

```
int key = TheKeyPressed( );
switch (key )
{
case 'u':
    SetPoints( A[0].z + Delta_z);
    break;
case 'd':
    SetPoints( A[0].z - Delta_z );
break;
}
```

Having started the animation, the z-coordinate of the points A[i] will be increased by Delta_z each time you press "**u**". Pressing "**d**" will decrease the z-coordinate.

Of course, we can also shade the two conoids. In `"folding_conoids2.cpp"` we introduced a new class derived from *RuledSurface*:

Listing from `"folding_conoids2.cpp"`:

```
class FoldingConoid: public RuledSurface
{
public:
   void Def( Color c, int m, int n, Real u1, Real u2,
             Real v1, Real v2, Real height, Real zeta )
   {
      H = height;
      Zeta = zeta;
      RuledSurface::Def( c, m, n, u1, u2, v1, v2 );
   };
   virtual Real FoldingFunction( Real u ) = NULL;
   virtual P3d DirectrixPoint( Real u )
   {
      return P3d( u, 0, Zeta );
   };
   virtual V3d DirectionVector( Real u )
   {
      Real a = FoldingFunction( u );
      Real b = H - a;
      Real z = ( a * a - b * b + ( H + Zeta ) * ( H - Zeta ) )
         / ( 2 * ( H - Zeta ) );
      Real y = sqrt( fabs( -z * z + 2 * Zeta * z +
                    a * a - Zeta * Zeta ) );
      return V3d( 0, y, z - Zeta );
   };
private:
   Real H;
   Real Zeta;
};
```

An instance of this class needs the standard input parameters of a parameterized surface, the total height of the initial rectangle, and the current z-coordinate zeta of the straight line a. The DirectrixPoint(...) and DirectionVector(...) functions of *RuledSurface* are replaced by the virtual function FoldingFunction(...). The computation of the direction vector is more or less a copy of the last example's SetPoints(...).

The first conoid of `"folding_conoids2.cpp"` is an instance of FoldingConoid, while the second is much easier to implement as an instance of *RuledSurface*:

Listing from "folding_conoids2.cpp":

```
class MyConoid1: public FoldingConoid
{
   virtual Real FoldingFunction( Real u )
   {
      return cos( 2 * PI / 5 * u ) + 5;
   }
};
MyConoid1 Conoid1;

class MyConoid2: public RuledSurface
{
   virtual P3d DirectrixPoint( Real u )
   {
      return P3d( u, 0, Height );
   };
   virtual V3d DirectionVector( Real u )
   {
      return Conoid1.SurfacePoint( u, 1 ) - DirectrixPoint( u );
   }
};
MyConoid2 Conoid2;
```

For the animation we use a pulsing real Z and define the conoids in Draw( ). Figure 3.50 shows a few screen shots from the program.

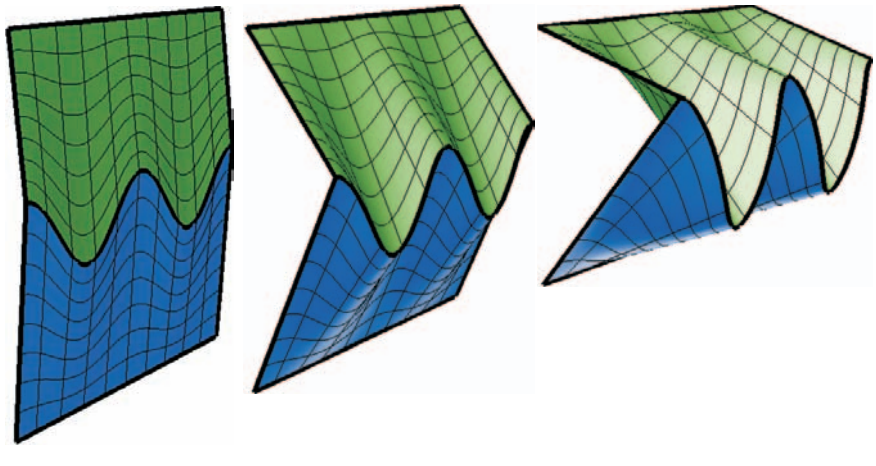

FIGURE 3.50. The folded conoids are smooth-shaded.

Listing from `"folding_conoids2.cpp"`:

```
Real z = Z.Next( );
Conoid1.Def( Blue, 50, 25, −5, 5, 0, 1, Height, z );
Conoid2.Def( Green, 50, 25, −5, 5, 0, 1 );
```

♢

In our next example, we go on folding surfaces. This time, however, we deal with torses rather than conoids.

Example 3.27. Folding torses
It is well known that the class of developable surfaces is rather small: Only cylinders, cones, and torses can be mapped isometrically to the plane. Developable surfaces are characterized by the existence of a one-parameter set of rulings along which the plane of tangency is constant. At least locally, a developable surface can be flattened — without being stretched or torn — to a plane area. Reversed, we can twist a plane area and get a torse. Have a look at `"folding_torses.cpp"` to see an animation. We fold and twist a circle ring and get a developable helical surface (a *helical torse*).

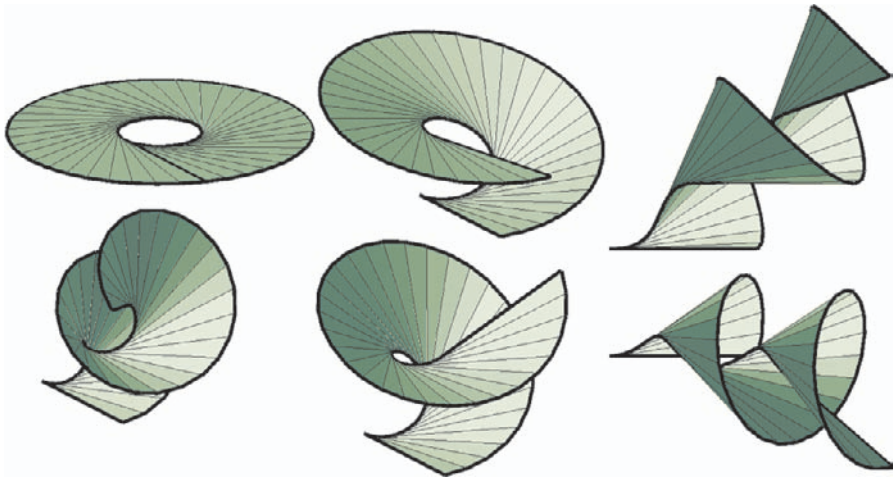

FIGURE 3.51. Twisting a flat ring to a screw torse.

The basic geometric idea of the program is very simple. The generators of the developed torse are the tangents of some circle c. We take a finite set $\{g_0, \ldots, g_n\}$ of equally distributed generators and rotate $g_1, \ldots, g_n$ about g_0 through a small

angle φ. Then we rotate $g_2, \ldots, g_n$ about g_1 through φ, and so on. After n steps, the generators g_i approximate a helical torse. We can start the whole thing again and, after n more steps, get another helical torse.

In `"folding_torses.cpp"` we implement a new class called TorseGrid. It consists of two sequences $A_0, \ldots, A_n$ and $B_0, \ldots, B_n$ of points. The points A_i and B_i determine a straight line $g_i = A_iB_i$ (the generator of the developed torse). We define some standard methods for defining, drawing, and shading the grid. The key method, however, is the following:

Listing from `"folding_torses.cpp"`:

```
void TorseGrid::Twist( int i, Real w )
{
    StrL3d a( A[i], B[i] );
    int j;
    for ( j = i + 1; j < PNum; j++ )
    {
        A[j].Rotate( a, w );
        B[j].Rotate( a, w );
    }
}
```

It performs just the rotation operation we have described above; i.e., the straight lines $g_{i+1}, \ldots, g_n$ are rotated around the axis g_i through w. In Init( ) we initialize the starting position with the help of two auxiliary functions (N is the total number of torse generators; L is their length):

Listing from `"folding_torses.cpp"`:

```
P3d P[N], Q[N];
int i;
Real t, delta = 2 * PI / ( N - 1 );
for ( i = 0, t = 0; i < N; i++, t += delta )
{
    P[i] = FirstPoint( t );
    Q[i] = SecondPoint( t, L );
}
Torse.Def( N, P, Q );
```

The auxiliary functions return a point on the circle c and on the corresponding circle tangent:

$$\begin{pmatrix} r\cos\mathsf{t} \\ r\sin\mathsf{t} \\ \zeta \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} r\cos\mathsf{t} - \mathsf{L}\sin\mathsf{t} \\ r\sin\mathsf{t} + \mathsf{L}\cos\mathsf{t} \\ \zeta \end{pmatrix}.$$

Now Draw( ) is really very simple. We shade the torse and twist it with each new frame:

Listing from `"folding_torses.cpp"`:

```
void Scene::Draw( )
{
  Torse.DrawGenerators( Black, THIN, 1e-2 );
  Torse.DrawBorder( Black, MEDIUM );
  Torse.ShadeFlat( Green );
  Torse.Twist( FrameNum( ) % N, Phi );
}
```

♢

Example 3.28. Minding's isometries
Surprisingly, the rather simple concept of Example 3.27 is capable of dealing with a very complex problem of differential geometry (of course, within certain limits that result from the fact that we use only a discrete number of generators). It is known as Minding's problem: *Bend a ruled surface* Φ *to another ruled surface* Φ^* *so that the generators of* Φ *and* Φ^* *correspond.*

All solutions to Minding's problem were described by E. Kruppa in 1951 ([20]). He could describe all surfaces that can be bent to a given ruled surface Φ in the Minding sense. Furthermore, he gave a criterion to decide whether two given surfaces Φ and Φ^* are Minding isometric (which is how two surfaces of that kind are called). This criterion is rather simple within the whole theory of Kruppa but far too complicated to be explained here. An immediate consequence of Kruppa's theory is the following proposition:

If two ruled surfaces Φ *and* Φ^* *are Minding isometric, their central lines necessarily correspond.*

This means that in a neighborhood of a ruling $r \subset \Phi$ the Minding isometric map is just a rotation about r. That is almost what we did in the previous example. The only (irrelevant) difference is that here successive generators are skew, while they intersect in `"helical_torse2.cpp"`. Hence, we can (at least locally) approximate any Minding isometric map.

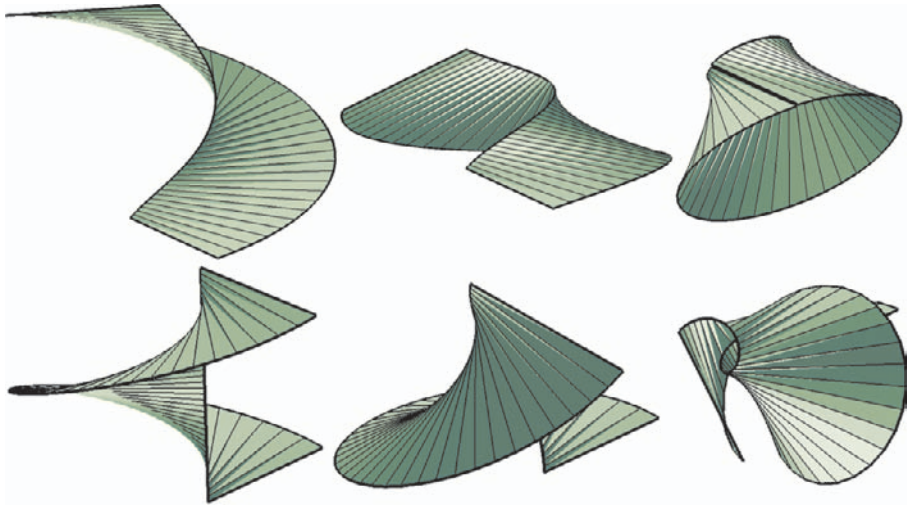

FIGURE 3.52. Bending helicoids in the Minding sense.

A first example is `"minding1.cpp"`. We added a few methods to the class TorseGrid and call it GeneratorGrid in this program. The additional methods are "getters" for A[i], B[i], and g_i = A[i]B[i] which are quite use for experimental usage of GeneratorGrid. A further special method is Shade(...):

Listing from `"minding1.cpp"`:

```
void GeneratorGrid::ShadeFlat( Color col )
{
   int i;

   Poly3d triangle1, triangle2;
   for ( i = 1; i < PNum; i++ )
   {
      triangle1.Def( col, 3, FILLED );
      triangle1[1] = A[i-1];
      triangle1[2] = A[i];
      triangle1[3] = B[i];
      triangle2.Def( col, 3, FILLED );
      triangle2[1] = A[i-1];
      triangle2[2] = B[i];
      triangle2[3] = B[i-1];
      triangle1.Shade( );
      triangle2.Shade( );
   }
}
```

We cannot shade a spatial quadrangle, so we shade two triangles instead. We already used this in "helical_torse2.cpp" even though it was not absolutely necessary, since we shaded only plane quadrangles.[20]

Furthermore, we changed the curves on which A[i] and B[i] are located:

Listing from "minding1.cpp":

```
P3d FirstPoint( Real t )
{
   return P3d( R * cos( t ), R * sin( t ), t * P );
}
P3d SecondPoint( Real t, Real dist )
{
   return FirstPoint( t ) + dist * V3d( -sin( t ), cos( t ), 0 );
}
```

That is, the initial surface is a helicoid with generators orthogonal but skew to the axis of rotation. The rest of the program can remain as it is, a good argument for the flexibility of our concept. Because of the symmetry of the initial configuration our animation is an exact model for the bending of a helical surface to a catenoid.

As an alternative, we can bend a right helicoid into a cone of revolution ("minding2.cpp"). This is, however, only a theoretical possibility, since it needs an infinite number of rotations. The cone of revolution is the result of a passage to the limit (Figure 3.52).

Just one more thing: In "minding1.cpp" we experimented a little with the parameters of the surface and the animation to produce a nice, closed catenoid. The animation will stop when B[0] and B[N] are sufficiently close. If you use other parameters, you will get a catenoid, too but there may be gaps or regions that are covered twice and the animation will continue. If the animation runs too long, the twisted surface becomes very long and small, and at some point the visibility algorithms will fail. ♢

In Section 2.1 we already talked about families of curves in two dimensions. Of course, there exist numerous 3D examples as well. We present two of them that concern curves on a torus.

[20]This method is not as sophisticated as Open Geometry's standard shading algorithms. When a surface triangle is to be shaded, Open Geometry creates the impression of a smooth surface by computing different colors for the three vertices and blending them.

Example 3.29. Spiric lines
In "torus_curves1.cpp" we intersect a torus T with a plane π parallel to its axis a. The resulting curve is called a *spiric line* and was already investigated by the ancient Greeks (PERSEUS, 130 B.C.). It is an algebraic curve of order four and can be of diverse shapes. Let A be the radius of the midcircle of T, B the radius of the meridians, and D the distance between π and a, see Figure 3.53.

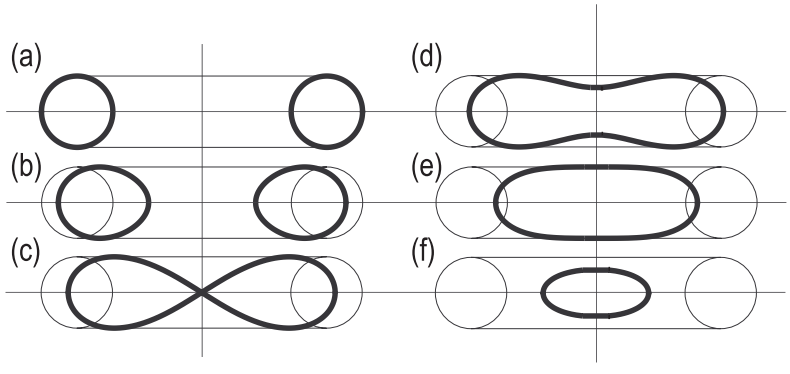

FIGURE 3.53. Different shapes of spiric lines. (a) $\mathsf{D} = 0$: two circles, (b) $0 < \mathsf{D} < \mathsf{A} - \mathsf{B}$: two ovals, (c) $\mathsf{D} = \mathsf{A} - \mathsf{B}$: lemniscate, (d) $\mathsf{A} - \mathsf{B} < \mathsf{D} < \mathsf{A}$: closed concave curve, (e) $\mathsf{A} = \mathsf{D}$: oval with flatpoints, (f) $\mathsf{A} < \mathsf{D} < \mathsf{A} + \mathsf{B}$: closed oval curve.

In the program we adjust a frame rectangle, representing the plane π, in a suitable position; i.e., in Init( ) we get it in a position parallel to the z-axis; in Draw( ) we translate it to the point $(0, \mathsf{y}, 0)^t$.

Listing from "torus_curves1.cpp":

```
Frame.Translate( 0, y − Frame[1].y, 0 );
```

The real number y is the current value of an instance of a pulsing real *PulsingReal* that swings harmonically between $\pm(\mathsf{A}+\mathsf{B})$. In the next step we shade the torus surface and intersect the carrier plane of the frame with the parallel circles of the torus:

Listing from "torus_curves1.cpp":

```
int i, n;
Circ3d circle;
Real v;
Real delta = PI / ( CircNum − 1 );
Plane p( Frame[1], Frame[2], Frame[3] );
```

```
P3d S[2];
for ( i = 0, v = 0; i <= CircNum; i++, v += delta )
{
   circle.Def( Red, P3d( 0, 0, B * sin( v ) ),
               V3d( 0, 0, 1 ), A - B * cos( v ), 10, EMPTY );
   n = circle.SectionWithPlane( p, S[0], S[1] );
   if ( n )
      Curve.AddPoint( S[0] );
}

Curve.Draw( MEDIUM );
Curve.Reflect( YZplane );
Curve.Draw( MEDIUM );
Curve.Reflect( XYplane );
Curve.Draw( MEDIUM );
Curve.Reflect( YZplane );
Curve.Draw( MEDIUM );
```

If there exist real intersection points, we add the first one to the *PathCurve3d* object Curve.[21] This path curve will store the point of one quarter of the spiric line. We draw it and reflect it three times at the coordinate planes to get the whole curve.

So far, so good. Our way of calculating the curve points leads to numerical problems if $A < D < A + B$. We need a correction in order to avoid gaps in the curve. Simply connecting the end points of corresponding quarters will help:

Listing from `"torus_curves1.cpp"`:

```
if ( fabs( y ) >= A - B )
{
   P3d H1 = Curve[1];
   P3d H2 = H1;
   H2.Reflect( YZplane );
   StraightLine3d( DarkBlue, H1, H2, MEDIUM );
   H1.Reflect( XYplane );
   H2.Reflect( XYplane );
   StraightLine3d( DarkBlue, H1, H2, MEDIUM );
}
```

[21] You see that the use of *PathCurve3d* is not limited to kinematic animations.

Now the output is entirely satisfactory, and our animation shows the transition between the different shapes of the spiric line (Figure 3.54).

Using the same idea, we can write a 2D animation as well (`"spiric_lines.cpp"`). There we compute the points of the spiric line directly and add them to our path curve. We used this program to produce Figure 3.53. Here are the relevant lines in Draw( ):

Listing from `"spiric_lines.cpp"`:

```
int i;
Real v;
Real delta = PI / ( PointNum - 1 );
for ( i = 0, v = 0; i <= PointNum; i++, v += delta )
{
   Real x = Sqr( A - B * cos( v ) ) - y * y;
   if ( x >= 0 )
   {
      SpiricLine.AddPoint( P2d( sqrt( x ), B * sin( v ) ) );
   }
}
```

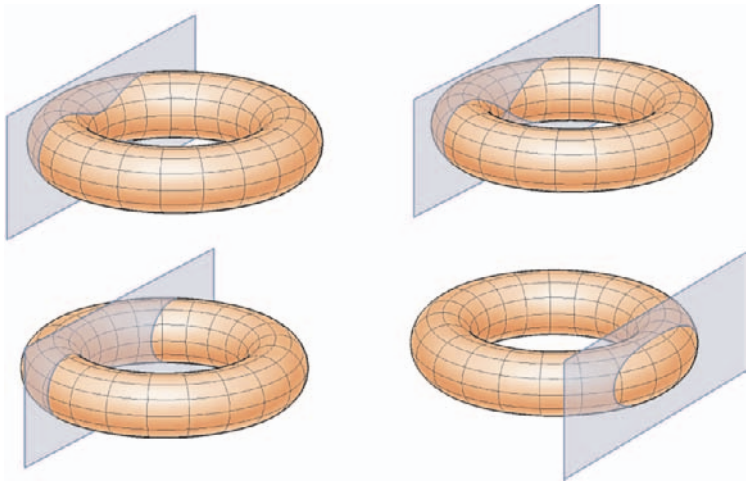

FIGURE 3.54. Spiric lines on a torus.

♢

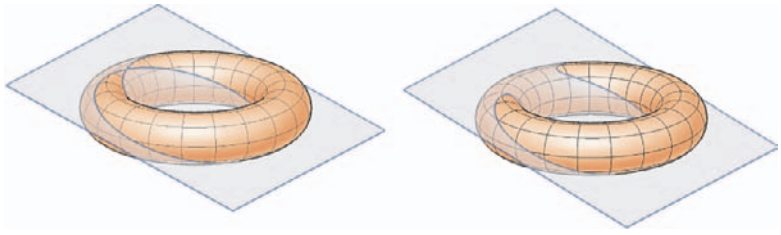

FIGURE 3.55. VILLARCEAU circles on a torus (left) and a generic intersection of torus and a plane through the torus center (right).

Example 3.30. Planar sections of a torus
A variation of the idea of Example 3.29 is to be found in `"torus_curves2.cpp"`. There we intersect a torus with a plane through its center. The resulting curves are again of order four and of diverse shapes.

The basic programming ideas are the same. We must, however, take care of a certain problem that arises when the intersecting plane is of small slope. Then only few parallel circles intersect the plane, and we do not get enough curve points.

Therefore, we replace the parallel circles by meridian circles if the angle Phi between the π and the $[x, y]$-plane is within a certain critical interval $[-\mathsf{Phi0}, \mathsf{Phi0}]$. By setting $\mathsf{Phi0} := \arcsin \mathsf{B}/\mathsf{A}$ we can ensure that all meridian circles have real intersection points with π. The relevant lines in Draw( ) read as follows:

Listing from `"torus_curves2.cpp"`:

```
if ( fabs( Phi ) < Phi0 )
{
   int i, j, n;
   Circ3d circle;
   Real u;
   Real delta = 2 * PI / ( CircNum - 1 );
   Plane p( Frame[1], Frame[2], Frame[3] );
   P3d S[2];
   for ( i = 0, u = 0; i <= CircNum; i++, u += delta )
   {
      circle.Def( Red, P3d( A * cos( u ), A * sin( u ), 0 ),
          V3d( -A * sin( u ), A * cos( u ), 0 ), B, 10, EMPTY );
      n = circle.SectionWithPlane( p, S[0], S[1] );
      for ( j = 0; j < n; j++ )
         Curve[j].AddPoint( S[j] );
   }
}
else
```

```
{
   int i, j, n;
   Circ3d circle;
   Real v;
   Real delta = 2 * PI / ( CircNum - 1 );
   Plane p( Frame[1], Frame[2], Frame[3] );
   P3d S[2];
   for ( i = 0, v = 0; i <= CircNum; i++, v += delta )
   {
      circle.Def( Red, P3d( 0, 0, B * sin( v ) ),
                  V3d( 0, 0, 1 ), A - B * cos( v ), 10, EMPTY );
      n = circle.SectionWithPlane( p, S[0], S[1] );
      for ( j = 0; j < n; j++ )
         Curve[j].AddPoint( S[j] );
   }
}
```

In Animate( ) we rotate the frame rectangle about the x-axis through a certain angle Delta. In addition, we update the rotation angle Phi that is used for the decision whether to use parallel circles or meridian circles for the calculation of curve points:

Listing from `"torus_curves2.cpp"`:

```
void Scene::Animate( )
{
   Frame.Rotate( Xaxis, Delta );
   Phi += Delta;
   if ( Phi > 90 )
      Phi -= 180;
}
```

By the way; in the case of Phi = ±Phi0 we get a pair of circles as intersection curve. This was discovered by VILLARCEAU in 1848. We choose Phi = −Phi0 as start value. So you can see the VILLARCEAU circles in the first frame (Figure 3.55). ♢

This page intentionally left blank

4

3D Graphics II

After having dealt with rather fundamental examples of 3D geometry in Chapter 3 we pass on to more advanced topics. You will find sections on

- spatial kinematics,
- complex animations
- Open Geometry's import and export facilities,
- Boolean operations,
- texture mapping.

We believe that even experienced Open Geometry programmers will find new ideas in this chapter. Novices may perhaps not understand every detail of the examples, but the results are definitely worth looking at.

4.1 Spatial Kinematics

The modeling of kinematic devices in three-space is usually much more demanding than the analogous task in 2D. Usually the involved motions are much more complicated than those of planar kinematics, and without a basic knowledge of the geometric concepts and a good store of methods for realizing them on the computer screen, it may be very difficult to visualize them. In the following we present some examples that will give a few hints on how to do this in Open Geometry 2.0.

Example 4.1. Escher kaleidocycles
First we present the OPEN GEOMETRY animation of a few sophisticated spatial mechanisms that are called ESCHER *kaleidocycles* in [33]. The name of the famous Dutch artist M.C. ESCHER is a bit misleading in this context; since [33] just makes use of ESCHER's well known regular plane-filling patterns (see [6]) as textures for beautiful little paper models. The mechanism itself was studied earlier by R. BRICARD.

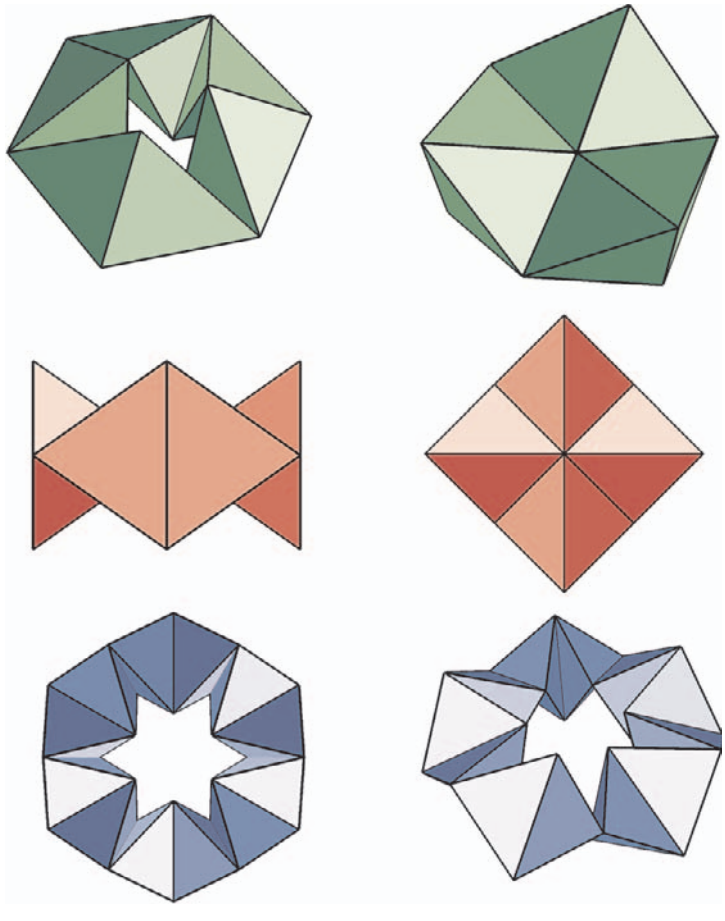

FIGURE 4.1. Various kaleidocycles (compare the programs `"kaleidocycle1.cpp"` and `"kaleidocycle2.cpp"`).

Three different types of these mechanisms are mentioned in [33]. In fact, using the same ideas, it is easy to derive infinitely many examples of kaleidocycles (we will do this in our programs). But before beginning to write any animation we have to understand their geometric properties. In Figure 4.1 several images are displayed. You can see that the objects consist of an even number n of congruent

tetrahedra. They are arranged in cyclic order. At corresponding edges they are linked in a way that a rotation about the edge is possible.

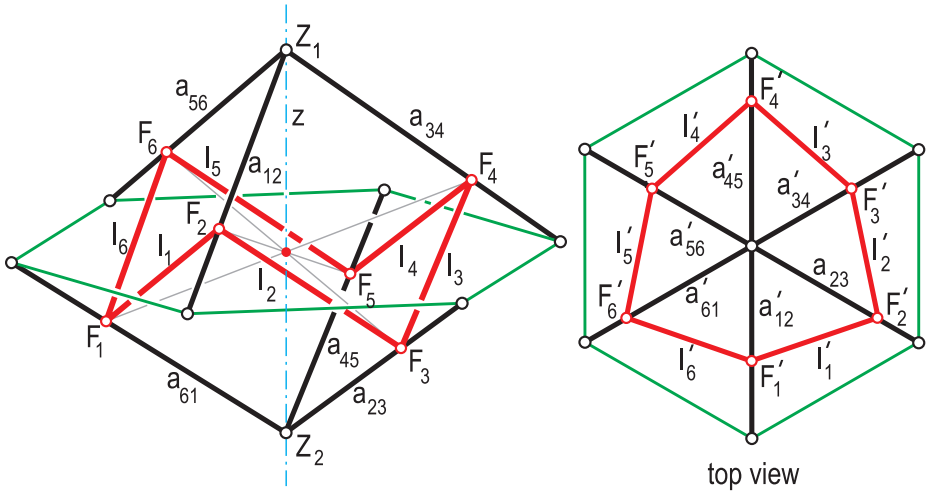

FIGURE 4.2. The cinematic mechanism of the Escher kaleidocycle.

For simplicity, we will restrict our attention to the case $n = 6$ in the following description. Then, from the point of view of spatial kinematics, we have six frames $\Sigma_1, \ldots, \Sigma_6$ (each represented by a tetrahedron) that are connected by six axes of revolution $a_{12}, \ldots, a_{61}$ (see Figure 4.2). A system of this kind is rigid in general, but, as in our example, it may turn out to be flexible under certain circumstances. A closer examination shows, for example, that two successive axes $a_{i-1,i}$ and $a_{i,i+1}$ have perpendicular directions.[1]

We can also represent the kaleidocycle by a mechanism M that consists of six straight lines $l_1, \ldots, l_6$ (l_i being the common normal of $a_{i-1,i}$ and $a_{i,i+1}$) that can be rotated about these axes.

Furthermore, we see that the distance d between the feet of l_i is constant. These conditions are already sufficient to guarantee the following properties of M (Figure 4.2):

1. The common normals $l_1, \ldots, l_6$ form a closed 3D polygon.
2. M possesses one-parameter mobility.

[1]In the following we will adopt the convention of always reading indices modulo 6. This gives us the possibility to write down our thoughts in both a readable and exact way.

3. Independent of the current position of M, the axes a_{12}, a_{34}, a_{56} and a_{23}, a_{45}, a_{61} intersect in a common point Z_1 or Z_2, respectively.[2]
4. The two opposite axes $a_{i,i+1}$ and $a_{i+3,i+4}$ span a plane σ_i, and M is always symmetric with respect to σ_i.
5. The three planes σ_1, σ_2, and σ_3 intersect in the straight line $z = [\, Z_1, Z_2 \,]$.
6. A rotation about z through an angle of $n \cdot 120^\circ$ brings σ_i to σ_{i+n}.

All of these properties are relevant to our attempt to realize this mechanism on the computer. In order to keep the symmetry during the animation, we will use a further frame Σ_0 in which the straight line z is fixed. Only translations in the direction of z will be permitted.

Now we have to do some precalculations to determine the position of our system's axes. We can assume that z is the third axis of a Cartesian coordinate system and that σ_0 is the $[x, z]$-plane. The axes a_{12} and a_{23} are then determined by their intersection points with z and the $[x, y]$-plane (compare Figure 4.3):

$$a_{12} \cap z = (0, 0, \mathsf{u})^t, \qquad a_{23} \cap z = (0, 0, \mathsf{u}^*)^t,$$
$$a_{12} \cap [\, x, y \,] = (\mathsf{v}, 0, 0)^t, \qquad a_{23} \cap [\, x, y \,] = (c\mathsf{v}^*, s\mathsf{v}^*, 0)^t,$$

where $c = \cos 60^\circ$ and $s = \sin 60^\circ$.

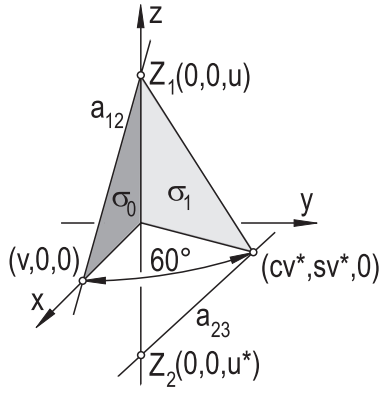

FIGURE 4.3. Calculating the position of the axes.

Taking into account the possible translation along z, we can, without loss of generality, assume that $\mathsf{v}^* = \mathsf{v}$. In the animation this will help us to keep the mechanism centered near the origin of our coordinate system.

The straight lines a_{12} and a_{23} have to meet the following two conditions:

[2] Of course, the points Z_1 and Z_2 vary as M is actuated.

- a_{12} and a_{23} are perpendicular.
- The feet F_1 and F_2 of their common normal l_1 are separated by the fixed distance d.

The first condition yields

$$\mathsf{u}^* = -\frac{c\mathsf{v}^2}{\mathsf{u}}.$$

After some strenuous computations, the second condition leads to the following relation between u and v:

$$c^2s^2\mathsf{v}^6 + c(2s^2\mathsf{u}^2 - cd^2)\mathsf{v}^4 + [s^2(\mathsf{u}^2 - d^2) - 2c^2d^2]\mathsf{u}^2\mathsf{v}^2 - d^2\mathsf{u}^4 = 0. \tag{1}$$

This equation reduces to an algebraic equation of degree three in the variable $\mathsf{V} := \mathsf{v}^2$. It can efficiently be solved by well-known formulas ([35]) that are integrated into Open Geometry 2.0. We will prefer this to solving equation (1) for u (which would require the solution to an equation of degree two only!) for the following reason:

We can prove that the algebraicalgebraic!equation of degree three that we get by substituting $\mathsf{v} = \sqrt{\mathsf{V}}$ always has *exactly* one positive solution $\mathsf{V}_0 > 0$ without any restriction on the parameter u. To prove this is not difficult, but still, this is not the right place for it. The solution V_0 is the only one that is useful for our purposes. This is quite convenient for our animation program. Solving (1) for u would mean that we have to take care of the range of parameter v to avoid imaginary solutions and the usual numerical problems. After all these considerations, we are ready to write the program (`"r6-mechanism.cpp"`).[3]

According to (1), we define a function CalcV(*Real* u) taking u as argument and returning v as value:

Listing from `"r6-mechanism.cpp"`:

```
Real CalcV( Real U )
{
    Real coeff[4];
    const Real u_2 = U * U;
    const Real d_2 = Dist * Dist;
    const Real angle_in_rad = 2 * PI / N;
    const Real c = cos( angle_in_rad );
    const Real c_2 = c * c;
    const Real s_2 = pow( sin( angle_in_rad ), 2 );
```

[3] *M* is called an *R6 mechanism* because it consists of six frames connected by 6 axes of Revolution.

```
    coeff[0] = -d_2 * u_2 * u_2;
    coeff[1] = u_2 * ( s_2 * ( u_2 - d_2 ) - 2 * c_2 * d_2 );
    coeff[2] = c * ( 2 * s_2 * u_2 - c * d_2 );
    coeff[3] = c_2 * s_2;
    Real u[3];
    int n = 0;
    n = ZerosOfPoly3( coeff, u );
    if ( n == 1 )
        return Sqrt( u[0] );
    else
        return Sqrt( u[2] );
}
```

Note that we need not worry much about finding the correct solution. The ZerosOfPoly3(...) function returns the real solutions in *ascending order*, and we know that there *exists exactly one positive root*. We use this function to compute the first axis axis[0] = a_{12}. Then we define the following axis axis[1] = a_{23} due to our considerations above and use the symmetry of M to get the rest. We draw the common normals, mark the relevant points, and the job is done![4]

Listing from `"r6-mechanism.cpp"`:

```
    Zaxis.LineDotted( Blue, -16, 16, 32, THIN );
    int i = 0;
    // Intersection point of the axes Axis[0],
    // Axis[2] and Axis[4] with the z-axis:
    P3d Z1( 0, 0, U );
    // Intersection point of the axes Axis[1],
    // Axis[3] and Axis[5] with the z-axis:
    P3d Z2( 0, 0, - cos( 2 * PI / N ) * V * V / U );

    // Intersection points of the axes and [xz]-plane
    // define the axes of revolution:
    P3d P[N];
    const Real angle = (Real) 360 / N;
    for ( i = 0; i < N; i += 2 )
    {
        P[i].Def( V, 0, 0 );
        P[i].Rotate( Zaxis, i * angle );
        Axis[i].Def( Z1, P[i] );
        StraightLine3d( Black, Z1, P[i], MEDIUM );
    }
```

[4] In the following listing the integer N denotes the number of axes, e.g., N = 6 in our case.

```
    for ( i = 1; i < N; i += 2 )
    {
        P[i].Def( V, 0, 0 );
        P[i].Rotate( Zaxis, i * angle );
        Axis[i].Def( Z2, P[i] );
        StraightLine3d( Black, Z2, P[i], MEDIUM );
    }

    // The feet of the common normals
    P3d F[N];
    V3d dummy; // Just needed as argument of CommonNormal(...)
    int j = 0;
    for ( i = 0; i < N; i++ )
    {
        j = (i+1) % N;  // Cyclic order of axes!
        Axis[i].CommonNormal( Axis[j], dummy, F[i], F[j] );
        StraightLine3d( Green, P[i], P[j], THIN );
        StraightLine3d( Red, F[i], F[j], THICK );
    }

    ...

    // Draw and mark the remaining relevant elements
    for ( i = 0; i < N; i++ )
    {
        StraightLine3d( Red, Origin, F[i], THIN );
        F[i].Mark( Red, 0.2, 0.1 );
        P[i].Mark( Black, 0.2, 0.1 );
    }
    Z1.Mark( Black, 0.2, 0.1 );
    Z2.Mark( Black, 0.2, 0.1 );
    Origin.Mark( Red, 0.1 );
}
```

For the animation we simply vary the point $\mathsf{Z1} = (0, 0, \mathsf{u})^t \in z$. We took a special distribution of the points to guarantee a more or less constant velocity of the animation. This ensures that the points $\mathsf{Z1}(\mathsf{u})$ vary on the whole z-axis but get slower in the region near the origin.

Listing from `"r6-mechanism.cpp"`:

```
void Scene::Animate( )
{
   // The following yields a more or less
   // constant animation velocity
   Phi += delta;
   U = tan( Phi ) * N / 3;
   // Calculate data for new position of axes
   V = CalcV( U );
}
```

Note that we took care, in our computations as well as in the OPEN GEOMETRY program, to avoid using the numerical values of $\cos 60°$ and $\sin 60°$. The reason for this is that the whole thing works just as fine with 8, 10, 12, or more axes of revolution! *You can use any even integer* $\mathsf{N} \geq 6$ *to build a kaleidocycle.* Try this by changing the global constant N at the beginning of `"r6-mechanism.cpp"` (in fact, the name of the program will then become obsolete as you simulate an RN-mechanism).

You might want to animate not only the abstract mechanism but also the charming models from [33]. Geometrically speaking, we have to do the following (compare `"kaleidocycle1.cpp"`): On each axis a_i we take two points A_i and B_i of fixed distance t_0 to the foot F_i of the common normal l_i of a_i and a_{i+1} (see Figure 4.4).

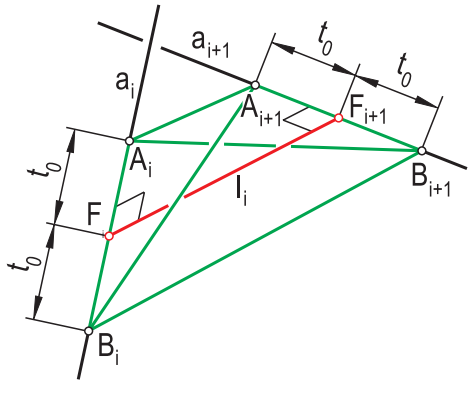

FIGURE 4.4. How to realize the physical model from [33]: The axes a_i and a_{i+1} are linked by the four edges of a tetrahedron.

Here t_0 is an arbitrary positive real number. It is, however, advisable for t_0 to satisfy the inequality

$$t_0 \leq t_0^{\max} = \frac{d}{\tan(2\pi/\mathsf{N})}$$

in order to avoid self intersections during the motion. The choice $t_0 = t_0^{\max}$ yields a mechanism where the hole in the center vanishes at one moment (Figure 4.1).

The points A_i, B_i, A_{i+1}, and B_{i+1} on two consecutive axes are the vertices of a tetrahedron (Figure 4.4) that was used in [33] to produce a physical model of the R6 mechanism. In OPEN GEOMETRY this can easily be imitated by introducing an array of 4N triangles (objects of type *Poly3d* with three vertices; Figure 4.1). Of course, you can increase the number of tetrahedra by changing the global constant N in `"kaleidocycle1.cpp"` as well.

We have not done it yet, but perhaps you want to equip the tetrahedra's faces with beautiful ESCHER textures. Have a look at Section 4.4 if you want to know how to do this. ♢

Our next example shows that really interesting geometry may be hidden in almost trivial things of everyday life.

Example 4.2. The peeling of an apple
In 1960 the Austrian geometer W. WUNDERLICH wrote a short paper with the astonishing title *"Geometrische Betrachtungen um eine Apfelschale"* (*"Geometric considerations concerning an apple-paring"*). He deals with certain geometric problems related to the peeling of an apple. We will explain this a little later after having exploited the beautiful kinematics of WUNDERLICH's paper for our purposes:

Let Γ be a cone of revolution with central angle 2α and apex O and let c be a circle (or rather a disk) with center O and radius R that is situated in a tangent plane of Γ. We want to study the motion induced by the rolling of c on Γ. It is the composition of two rotations about the axes of Γ and c. We will denote them by z and a, respectively.

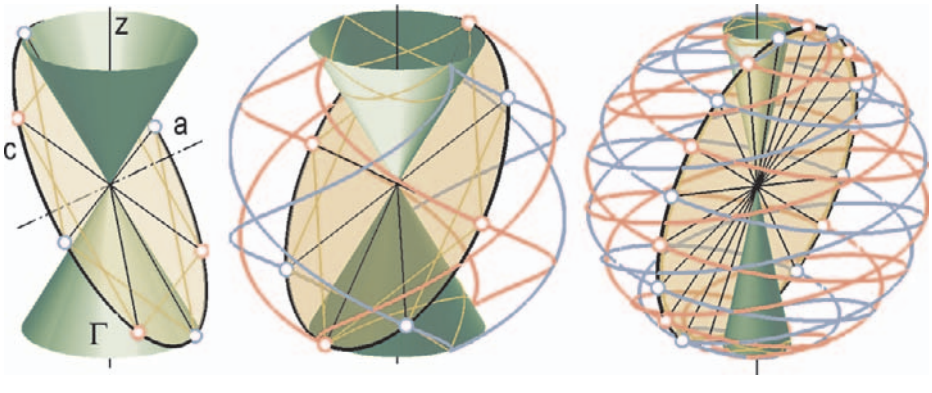

FIGURE 4.5. A circle c rolls on a cone of revolution Γ.

By Σ we denote the sphere with center O and radius R. It intersects Γ in two circles m and $\overline{m}$. During the motion, c rolls on both circles, m and $\overline{m}$. This gives

us the ratio $\omega_z : \omega_a$ of angular velocities we need for an animation. The circle m is of radius $r = R \sin\alpha$, and we obtain $\omega_z : \omega_a = R : r = 1 : \sin\alpha$.

In order to compute the position $P(\varphi)$ of a point P we have to do the following:

1. Rotate P and a around z through an angle of φ. This gives two new positions P_1 and a_1.
2. Rotate P_1 around a_1 through an angle of $\varphi \cdot \sin\alpha$.

That is exactly what we used in `"rolling_circle.cpp"`. However, in order to achieve a closed motion, we construct the parameters of our motion in a slightly different order:

Listing from `"rolling_circle.cpp"`:

```
const Real Radius = 8;
const int RotNumb = 4;
const Real Transmission = 2.0 / RotNumb;
const Real Alpha = asin( Transmission );
const Real Height = Radius * cos( Alpha );
const Real R = Radius * cos( PI * sin( Alpha ) );
```

Radius is the radius R of the rolling circle; RotNumb is the number of rotations about z that are necessary for a full period. Transmission gives the ratio of the angular velocities of the rotation around the fixed and rolling circles' axes, respectively. In order to get closed path curves it is sufficient (but not necessary) to compute it by dividing RotNumb by an integer value $n = 2.0$. Alpha, Height, and R are the parameters to determine the cone Γ.

What else do we need? We need two increments Delta and Phi for the rotation about z and a and a bunch of points and lines that are animated during the motion:

Listing from `"rolling_circle.cpp"`:

```
RegPoly3d Circle;

const int N = 2 * RotNumb;
P3d P[N+2];
Color Col[N];
PathCurve3d PathOfP[N];
StrL3d Generator[RotNumb];

StrL3d Axis, LineOfContact;

PathCurve3d Geodetic[N];
RegFrustum Cone1, Cone2;
```

Circle is the rolling circle; N is the number of trajectories we want to display. In addition, we will rotate N straight lines (generators of a certain ruled surface). They will trace a geodetic line on Γ, and we display it as well (*PathCurve3d* Geodetic[N]). Axis and LineOfContact are the axes of c and the generating line of Γ in the plane of c. The cone Γ itself will be a double cone; i.e., we use one regular frustum for each half of Γ. Now to their initialization:

Listing from `"rolling_circle.cpp"`:

```
void Scene::Init( )
{
   const Real r = Radius * sin( Alpha );
   Cone1.Def( Green, r, 0, Height, 100, HOLLOW, Zaxis );
   Cone2.Def( Green, r, 0, -Height, 100, HOLLOW, Zaxis );
   Cone1.Translate( 0, 0, -Height );
   Cone2.Translate( 0, 0, Height );

   V3d dir( 0, cos( Alpha ), sin( Alpha ) );
   Circle.Def( Black, Origin, dir, Radius, 100, FILLED );
   Axis = Circle.GetAxis( );

   int n = (int)(360 / Delta) + 2;
   int i;
   for ( i = 0; i < RotNumb; i++ )
   {
      Generator[i].Def( Origin, V3d( 0, -dir.z, dir.y ) );
      Generator[i].Rotate( Axis, i * 360 / N );
      P[i] = Generator[i].InBetweenPoint( Radius );
      Col[i] = i % 2 ? Red : Blue;
      PathOfP[i].Def( Col[i], n );
   }
   for ( i = RotNumb; i < N; i++ )
   {
      P[i] = Generator[i-RotNumb].InBetweenPoint( -Radius );
      Col[i] = i % 2 ? Red : Blue;
      PathOfP[i].Def( Col[i], n );
   }
   P[N] = P[0];
   P[N+1] = P[1];

   LineOfContact = Generator[0];

   for ( i = 0; i < N; i++ )
      Geodetic[i].Def( Yellow, 200 );
}
```

First, we define the cone Γ, the circle c, and its axis a. Then we define the generators and the path curves of certain special points. The integer N gives the minimal number of points on each path curve PathOfP[i]. The generators are equally distributed diameters of c; the points P[i] are their end points. Note that we insert two extra points P[N] and P[N + 1]. This will help us to imitate a cyclic order of the points P[i] in Draw( ).

Draw( ) itself is rather easy. We mark and draw the points and lines we previously defined. We add the points to the path curves and draw them as well. The only part of interest is the computation of the geodetic lines:

Listing from "rolling_circle.cpp":

```
StrL3d s;
P3d S;
const Real h = Height + 0.1;
const Real t = P[0].Distance( P[2] );

for ( i = 0; i < N; i++ )
{
    s.Def( P[i], P[i+2] );
    S = s * LineOfContact;
    if ( fabs( S.z ) < h )
        Geodetic[i].AddPoint( S );
    Geodetic[i].Draw( MEDIUM, 1e-3, 2 );
    s.Draw( Yellow, 0, t, MEDIUM );
}
```

We find a path point by intersecting the connecting line of two points P[i] and P[i + 2] (imitating cyclic order!) with the line of contact. We add this point to the path curve only if it lies on the part of Γ we are interested in. We can do this without problems, since the third parameter of Geodetic.Draw(...) hinders the drawing of line segments that are too long.

In Animate( ) we simply do our animation (rotations about z and a) as explained above. There is no need to print it here. Just a few words on the motion itself. From a geometric point of view, we display just *two* different trajectories and *two* different geodetics. However, all red and all blue points have the *same* trajectory. The trajectories (at least the parts between the cusps) are curves of constant slope on the sphere κ.

The connection to the peeling of an apple will become clear if you increase the number RotNumb of rotations. (Try it!) You can peel the apple along a spiral line without tearing the paring in two. Afterward, you can spread the paring on the table and you will get a double spiral in an "S" shape. Geometrically speaking, we can say that two consecutive points P[i] and P[i+1] determine the segment of a

straight line (the blade of your knife) that generates a torse T. The development of T yields the "S" shape. We visualize this in `"apple_paring.cpp"`.

The basic ideas are just the same as in `"rolling_circle.cpp"`: A circle rolls on a cone of revolution. However, there are quite a few differences in the drawing part: We display the paths of only two points. No straight lines are rotated. For this purpose we write two functions that compute the position of the circle axis and of a point P:

Listing from `"apple_paring.cpp"`:

```
StrL3d GetAxis( Real delta )
{
    StrL3d axis( Origin, V3d( 0, CosAlpha, SinAlpha ) );
    axis.Rotate( Zaxis, delta );
    return axis;
}
P3d GetPoint( Real delta )
{
    P3d P( 0, -Radius * SinAlpha, Radius * CosAlpha );
    P.Rotate( Zaxis, delta );
    P.Rotate( GetAxis( delta) , -delta * Transmission );
    return P;
}
```

The argument delta is the angle (in degrees) of rotation about z. We use these functions to define the paring torse T:

Listing from `"apple_paring.cpp"`:

```
class MyParingTorse: public RuledSurface
{
   virtual P3d DirectrixPoint( Real delta )
   {
      return GetPoint( delta );
   }
   virtual V3d DirectionVector( Real delta )
   {
      return V3d( GetPoint( delta ), GetPoint( delta - 360 ) );
   }
};
MyParingTorse ParingTorse;
```

The definition of the paring torse and the path of P in Init( ) is quite noteworthy:

Listing from "apple_paring.cpp":

```
int n = RotNumb − 4;
Real u0 = ( RotNumb − n ) ∗ 360;
ParingTorse.Def( Red, n ∗ 40, 2, u0, u0 + n ∗ 360, 0, 1 );
ParingTorse.PrepareContour( );
PointNum = (int)( 360 / Delta ∗ ( n + 1 ) + 2 );
PathOfP.Def( Black, PointNum );
```

We decide on an integer n that gives the number of windings of the torse that will be displayed. Using RotNumb-1 would yield almost the whole apple; we display less. Note that we use the relatively high number RotNumb = 12 in our program. Otherwise, the resulting image would not come close enough to an apple. Then we define the torse with an appropriate parameter interval. We use a rather high number of segments in the u-direction (the surface is rather "long"), while we calculate a minimum number of segments in v-direction. We can afford doing this even though we draw the contour of the surface: Since T is a torse, the v-lines are straight lines and the contour consists of *straight line segments*, too. The number of points on the path curve depends on the parameter interval of T and the increment Delta of the animation.

In Draw( ) we get the points of the cutting tool, mark them, and draw their path curve. (Remember: They actually have the same path curve!)

Listing from "apple_paring.cpp":

```
P3d P;
P = GetPoint( D );
P.Mark( Blue, 0.3, 0.2 );
GetPoint( D − 360 ).Mark( Blue, 0.3, 0.2 );
PathOfP.AddPoint( P );
PathOfP.Draw( THICK );
```

We draw the z-axis and shade the paring torse as well as the rolling circle. Finally, we rotate the circle about z through Delta and increase the global real D by Delta in Animate( ). The output of the program is displayed in Figure 4.6.

♢

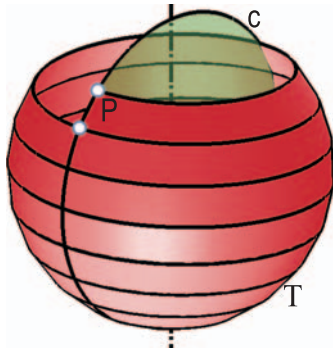

FIGURE 4.6. The peeling of an apple.

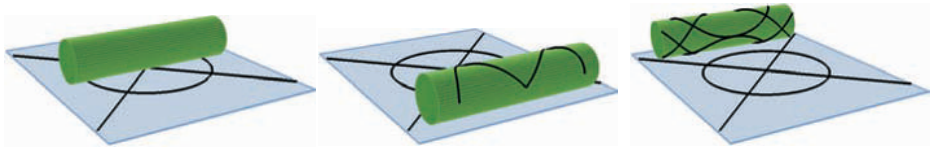

FIGURE 4.7. A circle and two straight lines print their image on a rolling cylinder.

Example 4.3. The anti development of a cylinder
A cylinder of revolution Γ is a developable surface. That is, you can cut it along a ruling and spread it into a plane π. This procedure maps a curve c^* on the cylinder to a curve c in the plane (development of c^*). Reversed, an arbitrary curve $c \subset \pi$ can be mapped to a curve c^* on the cylinder. We call c^* the *anti development of* c.

In `"rolling_cylinder.cpp"` we visualize the anti development of a circle c and two straight lines l_1 and l_2. Given the circle c and its supporting plane ε, we roll the cylinder on ε. Imagine now that the circle is freshly painted. It will leave a trace on Γ that yields exactly the anti developed image c^*. The same is true for l_1 and l_2.

Now, what are the essential code lines of `"rolling_cylinder.cpp"`? The rolling of Γ is not too difficult. We decide on its radius R1 and height Height (global constants) and define it in Init( ).

Listing from `"rolling_cylinder.cpp"`:

```
RollingCylinder.Def( Green, R1, Height, 50, SOLID, Yaxis );
RollingCylinder.Translate( 0, −0.5 ∗ Height, R1 );
Axis = RollingCylinder.GetAxis( );
XCoord = Axis.InBetweenPoint( 0 ).x;
```

Axis is the axis of Γ, XCoord is a real variable that stores the current x-value of Axis during the animation. Of course, we will roll the cylinder in the x-direction.

As animation parameter we use a pulsing real Phi. It gives the rotation amount along the Axis. Phi is defined as follows:

Listing from `"rolling_cylinder.cpp"`:

```
const Real a = 0.55 * R2;
Phi.Def( -a, 0.05, -a, a, HARMONIC );
```

Here R2 is the radius of c. This ensures that each point of c will be printed on the surface of Γ. Now let us have a quick look at Animate( ) before we return to the remaining parts of Init( ) and Draw( )

Listing from `"rolling_cylinder.cpp"`:

```
void Scene::Animate( )
{
  Real phi = Phi.Next( );
  Real delta = phi * Factor;
  XCoord += delta;

  Axis.Translate( delta, 0, 0 );
  Tau.Translate( V3d( delta, 0, 0 ) );
  RollingCylinder.Translate( delta, 0, 0 );
  Curve1.Translate( delta, 0, 0 );
  Curve2.Translate( delta, 0, 0 );
  Curve3.Translate( delta, 0, 0 );
  Curve4.Translate( delta, 0, 0 );

  RollingCylinder.Rotate( Axis, phi );
  Curve1.Rotate( Axis, phi );
  Curve2.Rotate( Axis, phi );
  Curve3.Rotate( Axis, phi );
  Curve4.Rotate( Axis, phi );
}
```

We translate all relevant elements by δ in direction of the x-axis. Rotation and translation are proportional; Factor gives just the right relation between angle of rotation and arc length on Γ. Note that XCoord is updated to the current x-value of the axis. We have already introduced the other players of `"rolling_cylinder.cpp"` in the above listing. They are the vertical plane Tau through the axis and four objects of type *PathCurve3d*. In Draw( ) they act as follows:

Listing from `"rolling_cylinder.cpp"`:

```
int n;
P3d S1, S2;
n = Circle.SectionWithPlane( Tau, S1, S2 );
if ( n )
   if ( S1.y > 0 )
   {
      Curve1.AddPoint( S1 );
      Curve2.AddPoint( S2 );
   }
   else
   {
      Curve1.AddPoint( S2 );
      Curve2.AddPoint( S1 );
   }

Tau.SectionWithStraightLine( Line3, S1, Dummy );
if ( fabs( S1.y ) < H2 )
   Curve3.AddPoint( S1 );

Tau.SectionWithStraightLine( Line4, S1, Dummy );
if ( fabs( S1.y ) < H2 )
   Curve4.AddPoint( S1 );
```

We intersect Tau with the circle c and add the intersection points to Curve1 and Curve2 according to the sign of their y-coordinates. We do the same with the straight lines l_1 and l_2. Since we do not want to paint the air, we add the intersection points only if they are within a relevant y-interval.

There is just one little thing to do. The image c^* of c is obviously a closed curve. However, we draw two half curves, and our method is not capable of adding the transition points to c^*. Thus, we have to add them manually. We use two global constants and two global variables for this purpose

Listing from `"rolling_cylinder.cpp"`:

```
const P3d P1( R2, 0, 0 );
const P3d P2( -R2, 0, 0 );
Boolean AddP1 = true, AddP2 = true;
```

Here P1 and P2 are the critical points. We add them to the curves in Draw( ):

Listing from `"rolling_cylinder.cpp"`:

```
if ( XCoord > R2 && AddP1 )
{
   Curve1.AddPoint( P1 );
   Curve2.AddPoint( P1 );
   AddP1 = false;
}
if ( XCoord < -R2 && AddP2 )
{
   Curve1.AddPoint( P2 );
   Curve2.AddPoint( P2 );
   AddP2 = false;
}
```

Now everything is fine. We add a rectangular frame to the scene and watch the cylinder roll (Figure 4.7). In this picture the diameter of c is just the arc length of a cylinder circle. Thus, the anti developed image has a double tangent with identical points of tangency. ♢

Example 4.4. Rolling cone
Of course, we can also roll a cone Γ on a plane and paint the image of a circle on it (Figure 4.8). We did this in `"rolling_cone.cpp"`. The basic ideas are the same as in `"rolling_cylinder.cpp"`. There are, however, certain important differences.

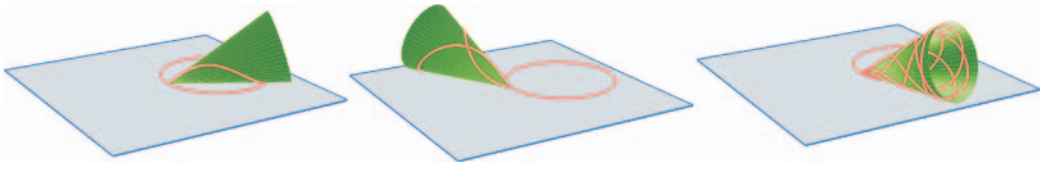

FIGURE 4.8. The anti development of a circle on a cone of revolution.

First of all, there is no need to roll the cone backwards and forwards. During the motion the apex O of Γ remains fixed. Thus, Γ will stay in a limited area all the time. We denote the height and radius of the base circle by h and r, respectively. Then the generators are of length $e = \sqrt{h^2 + r^2}$. The motion is periodic only if $r : e$ is rational, i.e., in general, it is not.

We take O as the origin of a Cartesian coordinate system. The fixed plane will be $[x, y]$. Then the motion is the composition of a rotation about z and about the axis a of Γ. The ratio of the corresponding angular velocities φ and ϱ is $r : e$. We define the cone and its axis in Init( ):

Listing from "rolling_cone.cpp":

```
RollingCone.Def( Green, R1, Height, 50, HOLLOW, Zaxis );
RollingCone.Translate( 0, 0, -Height );
RollingCone.Rotate( Xaxis, 90 + Deg( atan( R1 / Height ) ) );
Axis = RollingCone.GetAxis( );
```

The global constants R1 and Height have been defined thus:

Listing from "rolling_cone.cpp":

```
const Real R1 = 3, Side = 3.25 * R1;
const Real Height = Sqrt( Side * Side - R1 * R1 );
```

Thus, the ratio of radius and length of generator is rational and the motion will be periodic.

We define a circle c as in the previous program. It will print its image on the cone surface during the animation. In addition to "rolling_cylinder.cpp", we use a global constant ApexInside of type *Boolean*. It tells us whether the apex O lies inside the circle or not. We need it in Draw( ) in order to determine the correct points to the path curves.

We determine the curve points in the following way: The current line of contact between the cone and the base plane is stored in a global variable StrL. We intersect StrL with the circle. This is realized by intersecting c with an auxiliary plane τ through the line of contact and parallel to the z-axis.

Now we have to decide which intersection points we add to which path curves. If the apex lies inside c, we add the point with positive parameter value on StrL to Curve1. Curve2 remains empty. If the apex is outside c, we add the points only if they have positive parameter values.

Furthermore, we have to take into account two critical points P1 and P2 that have to be added when StrL is tangent to c. We solve this problem by testing the distance from P1 and P2 to the auxiliary plane τ. If it is below a certain limit Tol, we add the points to the curve.[5] The complete code of this passage reads as follows:

Listing from `"rolling_cone.cpp"`:

```
int n;
P3d S1, S2;
Plane tau( P3d( 0, 0, 1 ), StrL );
n = Circle.SectionWithPlane( tau, S1, S2 );
if ( ApexInside )
{
   if ( StrL.GetParameter( S1 ) > 0 )
      Curve1.AddPoint( S1 );
   else
      Curve1.AddPoint( S2 );
}
else
{
   if ( P1.IsInPlane( tau, Tol ) && StrL.GetParameter( P1 ) > 0 )
   {
      Curve1.AddPoint( P1 );
      Curve2.AddPoint( P1 );
   }
   if ( P2.IsInPlane( tau, Tol ) && StrL.GetParameter( P2 ) > 0 )
   {
      Curve1.AddPoint( P2 );
      Curve2.AddPoint( P2 );
   }
   if ( n )
   {
      Real s1 = StrL.GetParameter( S1 );
      if ( s1 > 0.02 )
      {
         if ( s1 > StrL.GetParameter( S2 ) )
         {
            Curve1.AddPoint( S1 );
            Curve2.AddPoint( S2 );
         }
         else
         {
```

[5] This is a little dirty. It may happen that we add them twice or even more often. But since the distances in this area of the curve are too small, you will never be able to see this little mistake.

```
            Curve1.AddPoint( S2 );
            Curve2.AddPoint( S1 );
        }
    }
  }
}
```

Finally, in Animate( ) we replace the translation of `"rolling_cylinder.cpp"` by a rotation about z:

Listing from `"rolling_cone.cpp"`:

```
void Scene::Animate( )
{
  Axis.Rotate( Zaxis, Phi );
  StrL.Rotate( Zaxis, Phi );
  RollingCone.Rotate( Zaxis, Phi );
  Curve1.Rotate( Zaxis, Phi );
  Curve2.Rotate( Zaxis, Phi );

  RollingCone.Rotate( Axis, Rho );
  Curve1.Rotate( Axis, Rho );
  Curve2.Rotate( Axis, Rho );
}
```

♢

Example 4.5. Rolling hyperboloids
In the preceding examples we have already presented a few rolling surfaces: cones, cylinders, and planes rolling on one another. These were all developable surfaces; i.e., along one generating line the plane of tangency is constant. There exist, however, nontrivial rolling motions of general (nondevelopable) ruled surfaces. Consider the example of `"rolling_hyperboloids.cpp"`. There, two congruent hyperboloids roll on one another. This motion has technical relevance. If we equip the hyperboloids with gears, they can transmit a uniform rotation between their axis.[6]

We define the axes of the hyperboloids Hyp1 and Hyp2 as follows:

[6] Actually, the motion is not a pure rolling. There is a translation component as well. This causes stringent requirements on the material, and therefore, gear wheels of that kind are not too common in practice.

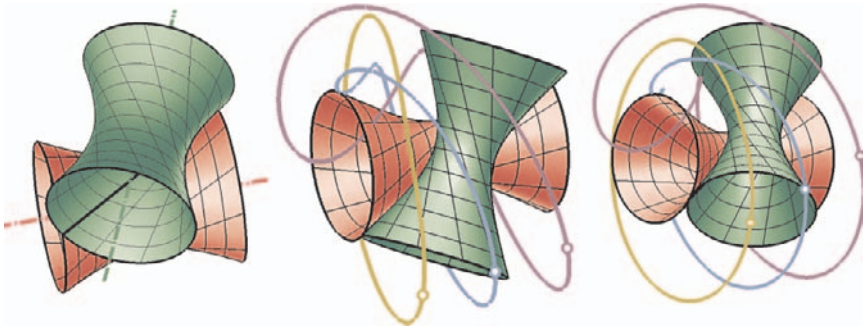

FIGURE 4.9. Two hyperboloids of revolution roll on one another (left). Three path curves of the relative motion with fixed Hyp2.

Listing from `"rolling_hyperboloids.cpp"`:

```
Axis1.Def( P3d( 0, 0,  Rad ), V3d( cos( Angle ),
                sin( Angle ), 0 ) );
Axis2.Def( P3d( 0, 0, −Rad ), V3d( cos( Angle ),
                −sin( Angle ), 0 ) );
```

Rad and Angle are global constants to determine the shape of the hyperboloid. The hyperboloids themselves are parameterized surfaces (the second hyperboloid is defined analogously):

Listing from `"rolling_hyperboloids.cpp"`:

```
class MyHyperboloid1: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real u, Real v )
   {
      P3d P( u, 0, Eps );
      P.Rotate( Axis1, v );
      return P;
   }
   StrL3d Ruling( Real v )
   {
      return StrL3d( SurfacePoint( 0, v ), SurfacePoint( 1, v ) );
   }
};
MyHyperboloid1 Hyp1;
```

Hyp1 and Hyp2 touch along the x-axis of our coordinate system. Because of this, we need a small positive real Eps to avoid visibility problems along the line of contact. Be aware that Angle must be chosen with care. The intersection curve of Hyp1 and Hyp2 is of order 4. In our case it splits into one double line (the x-axis) and a remaining part of second order that needs to be imaginary. In Animate( ) we rotate the hyperboloids about their respective axes in opposite directions. The output of the program can be seen in Figure 4.9.

Listing from "rolling_hyperboloids.cpp":

```
Hyp1.Rotate( Axis1, Speed );
Hyp2.Rotate( Axis2, -Speed );
```

In "rolling_hyperboloid.cpp" (singular!) we literally take a slightly different point of view: Again, we use two hyperboloids Hyp1 and Hyp2 that initially touch along the x-axis, but we connect the fixed coordinate frame with Hyp2. Thus, we get a rather curious motion where Hyp1 rolls on Hyp2. The Init( ) and Draw( ) parts of the program hardly change. In addition to "rolling_hyperboloids.cpp", we trace the paths of three points P, Q, and R, and we draw the line of contact at any moment. The most important change takes place in Animate( ). We rotate everything about the axis of Hyp2 and then, if necessary, about the axis of Hyp1 (compare Figure 4.9).

Listing from "rolling_hyperboloid.cpp":

```
void Scene::Animate( )
{
   Axis1.Rotate( Axis2, Speed );
   LineOfContact.Rotate( Axis2, Speed );
   Hyp1.Rotate( Axis2, Speed );
   P.Rotate( Axis2, Speed );
   Q.Rotate( Axis2, Speed );
   R.Rotate( Axis2, Speed );

   Hyp1.Rotate( Axis1, Speed );
   P.Rotate( Axis1, Speed );
   Q.Rotate( Axis1, Speed );
   R.Rotate( Axis1, Speed );
}
```

♢

Example 4.6. Kardan joint
In technical applications it is often necessary to transmit a rotation from an axis a to a second axis b of different direction. Think of a car's steering wheel: By turning it around, you activate the connecting axis of the tires. There exist different joints that have been developed to perform tasks of this kind. In this example we present an animation for one of them: the *kardan joint*.

It can be used for the transmission of a rotation between two intersecting axes. Have a look at Figure 4.10: The two axes a and b end in shafts that are attached to a cross. The points$A$ and B are end points of the cross arms. The shafts can still rotate about the cross arms.

If the kinematics are not completely clear to you, we recommend that you start the corresponding OPEN GEOMETRY animation in `"kardan.cpp"`. At start-up, the program displays the wire-frame model. You can start the animation and study it. If you want to see the solid model, you have to restart the program (press <`Ctrl + Shift + R`> or use the menu item "Program→Restart Program").

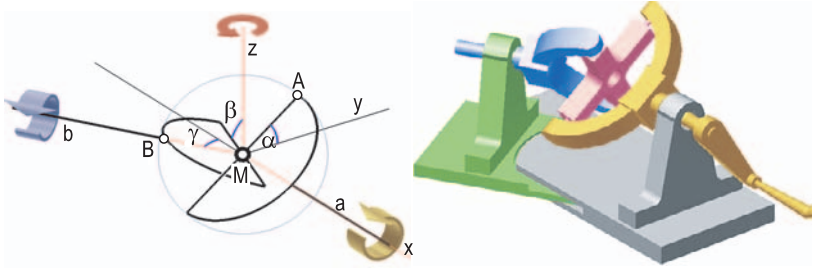

FIGURE 4.10. A wire frame model and a solid model of a kardan joint.

Note that the angle γ between a and b will be increased by some increment after one full turn. If γ is large enough, you can clearly verify that the transmission is not uniform. The axis a revolves with constant angular velocity, while the motion of b is irregular. This behavior limits the technical applications of kardan joints, especially for the transmission of high-speed rotations.

However, for our computer animation this is not the main problem. It is not too difficult to see that the respective angles of rotation α and β are related via the equation

$$\tan\beta = \frac{1}{\cos\gamma}\tan\alpha \tag{2}$$

(compare Figure 4.10). To us, the main problem is the motion of the cross in the center of the joint. For the animation in our sample file `"kardan.cpp"` we proceed as follows:

1. We rotate the axis a (and all related objects) by a constant angle increment, compute the corresponding angle of rotation β of b from equation (2) and rotate b and all related objects to their correct positions.

2. We rotate the middle cross *back to some initial position*. This makes the next step much easier.

3. We consider two end points A and B of the current cross position and the corresponding points A_0 and B_0 of the initial cross position and determine the unique rotation R that brings A_0 to A and B_0 to B.

4. Since the cross center M remains fixed during the motion, we can bring the cross into the current position by applying the rotation R to the cross in initial position.

The idea of repeatedly rotating an object back to some initial position is sometimes useful for kinematic animations. We used it in Example 2.17 as well. The axis of rotation is, of course, found by intersecting the planes of symmetry of two pairs of corresponding points.

Now we come to the description of the OPEN GEOMETRY implementation. We use the following global variables:

Listing from `"kardan.cpp"`:

```
Real Alpha, Beta, Gamma, Omega, Omega0, SG, CG;
P3d M, A, B, A0, B0;
StrL3d Axis[3], MomAxis, MomAxis0;
   // axes of object and for rotating the middle cross

const Real Rad = 3.3; // radius of shafts
Cad3Data Obj; // CAD3D-data for solid model

Boolean WireFrame = 0; // show wire frame or solid model
Arrow3d RotArrow1, RotArrow2, RotArrow3; // indicate rotation of axes
```

The first few variables (Alpha, Beta, Gamma, A, B) refer to the geometric description of the mechanism as given above; A0 and B0 are cross points in zero position; in Omega and Omega0 the angle of rotation from zero position to the current position will be stored. The corresponding axes are MomAxis and MomAxis0, respectively. We need two variables for that in order to undo the last rotation before performing the current one. SG and SC are abbreviations for the sine and cosine of the current axis angle $\gamma =$ Gamma. Furthermore, we use $a =$ Axis[0], $b =$ Axis[1], and $z =$ Axis[2] (the z-axis of our coordinate system; it is perpendicular to a and b). The remaining variables are needed for the drawing of the scene.

In Init( ) we initialize the starting configuration of the mechanism. The function do_calcs( ) computes some of the relevant data. It is separated from Init( ) because the user is not expected to change anything in that part.

Listing from `"kardan.cpp"`:

```
void do_calcs( )
{
   M = Obj[0][0].GetCenterOfBoundingBox( );
   M.z = 4.5;

   Axis[0].Def( M, Zdir );
   Axis[1].Def( M, Xdir );
   Axis[2].Def( M, Xdir );
   Axis[2].Rotate( Axis[0], Gamma - 180 );

   A = M + Rad*Ydir;
   B = M + Rad*Zdir;
   A0 = A;
   B0 = B;

   Real g = Arc( Gamma );
   SG = sin( g );
   CG = cos( g );
   Omega0 = 0;
   MomAxis0 = Zaxis;
}
void Scene::Init( )
{
   WireFrame = Boolean( 1 - WireFrame );

   Obj.Delete( );
   Obj.Def( 100 );

   Alpha = Beta = 0;
   Gamma = 30;

   Color col[6] = { Gray, Green, Pink, Yellow, Yellow, Blue};
   Obj.ReadNewMember( "DATA/LLZ/kardan_complete.llz", 0.1 );
   Obj[0].SetColors( col );

   do_calcs( );

   if ( WireFrame )
      Write( "This is..." );
   AllowRestart( );

   RotArrow1.Def( 3, Axis[0], 6, V3d( 2,2,1 ) );
   RotArrow2.Def( 4, Axis[1], 10, V3d( 2,2,5 ) );
   RotArrow3.Def( -4, Axis[2], 10, V3d( 2,2,5 ) );
}
```

Draw( ) is very simple. In fact, we do nothing but call a draw routine for the wire frame model or shade all CAD3D objects. We do not list the draw routine; it is rather lengthy and of little interest.

Listing from "kardan.cpp":

```
void Scene::Draw( )
{
  if ( WireFrame )
    ShowRotations( );
  else
    Obj.Shade( );
}
```

The really important part of "kardan.cpp" is Animate( ). We display it together with a few comments. With the help of the basic idea (rotating the middle cross back and forth) it is not too difficult to understand the code:

Listing from "kardan.cpp":

```
void Scene::Animate( )
{
  Alpha += 3; // increment angle of rotation

  // adjust arrows
  RotArrow2.Twist( 3 );
  RotArrow3.Twist( -3 );

  // calculate current angles and points
  Real a = Arc( Alpha );
  Real Beta0 = Beta;
  Real b = ArcTan2( Tan( a ), CG );
  Beta = Deg( b );
  Beta -= 180;
  while ( fabs( Beta - Alpha ) > 45 )
  {
    Beta += 180;
  }
  b = Arc( Beta );
  Real sb = sin( b );
  V3d MA( 0, cos( a ), sin ( a ));
  V3d MB( SG * sb, -CG * sb, cos ( b ));
  A = M + Rad * MA;
  B = M + Rad * MB;
```

```
// In order to adjust the middle cross, we must rotate
// the rod A0A into the position B0B. The axis of rotation
// (MomAxis) is obtained by intersecting two planes of
// symmetry:
Rod3d A0A( Gray, A0, A );
Rod3d B0B( Gray, B0, B );
Plane s1( A0A ), s2( B0B );
MomAxis = s1 * s2;
MomAxis.SetPoint( M );

// Now we have to compute the angle of rotation:
P3d N = MomAxis.NormalProjectionOfPoint( A );
V3d v = A - N;
v.Normalize( );
V3d v0 = A0 - N;
v0.Normalize( );
Omega = Deg( ArcCos( v * v0 ) );

// Change sign of Omega if necessary:
P3d A1 = A0;
A1.Rotate( MomAxis, Omega );
if ( A1.Distance( A ) > 1e-3 )
{
   Omega *= -1;
   A1 = A0;
   A1.Rotate( MomAxis, Omega );
   if ( A1.Distance( A ) > 1e-3 )
      SafeExit( "problems" );
}

P3d B1 = B0;
B1.Rotate( MomAxis, Omega );
if ( B1.Distance( B ) > 1e-3 )
{
   ShowReal( "B dist =", B.Distance( B1 ) );
      // Should not happen!
}

// Rotate all objects:
if ( Omega0 )
   Obj[0][2].Rotate( MomAxis0, -Omega0 );
Obj[0][2].Rotate( MomAxis, Omega );
Obj[0].Rotate( 3, 4, Axis[1], 3 );
Obj[0][5].Rotate( Axis[2], Beta0 - Beta );
Omega0 = Omega;
MomAxis0 = MomAxis;

// Increase angle between two main axes of
// revolution by 4 after one full turn.
```

```
    if ( fabs( Alpha − 360 ) < 0.01)
    {
       change_gamma( 4 );
    }
}
```

Animate( ) calls the function change_gamma(...) after each full rotation. There, the angle γ between the axes a and b is increased. Consequently, some of the global variables have to be adapted:

Listing from `"kardan.cpp"`:

```
void change_gamma( Real delta )
{
   Gamma += delta;
   Axis[2].Rotate( Axis[0], delta );
   Obj[0][1].Rotate( Axis[0], delta );
   Obj[0][5].Rotate( Axis[0], delta );
   RotArrow3.Rotate( Axis[0], delta );
   Real g = Arc( Gamma );
   SG = sin( g );
   CG = cos( g );
   Alpha = 0;
   Omega0 = 0;
   MomAxis0 = Zaxis;
}
```

For a picture of the final output, please, see Figure 4.10. ♢

4.2 Import and Export

The import and export facilities of Open Geometry have been considerably enlarged in version 2.0. We still support import of ASC files and Cad3D objects. The latter can be smooth-shaded and equipped with different colors in Open Geometry 2.0. In the near future, we are planning to implement the import of ASE files as well.

Open Geometry 2.0 exportation to DXF, BMP, EPS files and POV-Ray include files (new!) is possible. We implemented the EPS export option for 3D scenes as well (new!). Of course, it can work only with certain drawbacks, since PostScript does not support z-buffering. Later in this chapter we will explain this more closely.

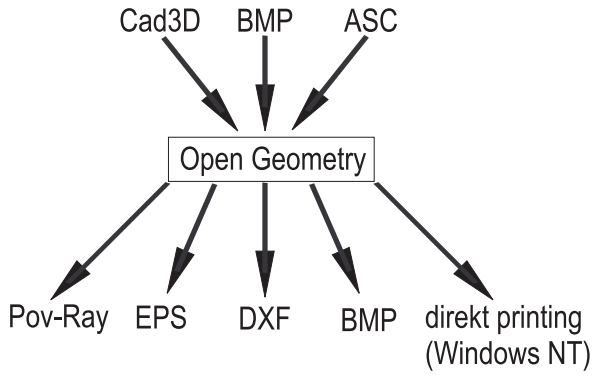

FIGURE 4.11. The import and export facilities of OPEN GEOMETRY 2.0.

The graphic in Figure 4.11 illustrates the import and export facilities of OPEN GEOMETRY 2.0.

We are especially fond of the new export possibility in POV-Ray's *Scene description language*. POV-Ray is a freeware ray-tracing program. With only a little effort you can create photorealistic renderings of your OPEN GEOMETRY scenes. Many pictures in this book have been created that way.

The import of ASC files is explained in Section 11.2 of [14]. Examples 3.3 and 4.6 show how to integrate CAD3D objects in OPEN GEOMETRY. See also Figure 4.15 and Figure 4.16.

In the following we will explain the different export facilities of OPEN GEOMETRY in more detail.

BMP

OPEN GEOMETRY can export the content of your screen to BMP files. This may be an alternative to making a hard copy of your screen as described in [14], pp. 320 ff. Furthermore, it is really useful for the preparation of animated GIF or AVI files.

The export of one single bitmap image is very easy: Simply choose "Image→Export Image→Save Image as BMP" from the OPEN GEOMETRY menu. You can choose between 8-bit and 24-bit color resolution.[7] Alternatively, you can use the shortcuts <`Ctrl + B`> and <`Ctrl + 8`>. Note that the final number of pixels depends on the size of your OPEN GEOMETRY window as well as on your screen resolution.

[7]If you choose 8-bit resolution, make sure that the OPEN GEOMETRY window side lengths are multiples of eight. Otherwise, the output will not be correct. You can adjust the window size via the menu item "Image→Window→Window dimensions".

There exists a third way of exporting in BMP format: Use the method SaveAsBitmapAnimation(...) of the class *Scene*. Its header can be found in "scene.h":

```
int SaveAsBitmapAnimation( const char *path = "BMP",
                           int start_frame = 1,
                           int increase = 1,
                           int max_files = 50,
                           int bit_depth = 24 );
```

It has to be put in the Animate( ) part. The first argument specifies the path for you pictures to be stored. The default directory for this is specified in "og.ini". That is, if you work with the default settings, they will be written to "OPENGEOM/BMP/". During the animation the frames with numbers

$$\text{start\_frame},\ \text{start\_frame} + 1 \cdot \text{increase}, \ldots, \text{start\_frame} + \text{n} \cdot \text{increase}$$

will be stored as "001.bmp", "002.bmp", "003.bmp" until the number max_file is reached. Be careful not to use up all your free disk space while doing this: Scaling the Open Geometry window to a reasonable size and minimizing white space might help a lot.

Having created a series of bitmaps, you can proceed to convert them into GIF files and arrange animations for the Internet. Have a look at our home page to see examples of this (http://www.uni-ak.ac.at/opengeom/).

DXF

Exporting DXF (Drawing eXchange Format) is very easy in Open Geometry, and nearly every CAD program can load such files. You can store your 3D data as DXF file by simply selecting "Image→Export Image as DXF file".

There are a few things to keep in mind, however, when working with DXF files.

- DXF files that contain many polygons need a lot of disk space.
- You cannot store mapping coordinates or material properties in DXF files.
- DXF files are text files that cannot be loaded very quickly.
- Open Geometry will store only polygons. It will not export lines!

If you are going to load DXF files with 3D Studio Max, activate the "force two sided" option and keep in mind that Open Geometry stores the whole scene as a single object.

If you want to create several objects, you can proceed as follows:

- Let Open Geometry draw only the part you want to store as one object (by commenting out the other command lines).
- Save the scene under a distinct name.
- Repeat the process for the other parts of the scene. Since the storage is done in 3D, you need not worry about viewport or camera.

If you want to know more about Open Geometry and DXF, have a look at [14], pp. 307 ff. There, you will find a few sample programs for the import and export of DXF files with the help of Open Geometry.

EPS

Open Geometry allows you to export your scenes as EPS files (Encapsulated Post Script). The EPS file format is not platform-specific and is vector-oriented. The output quality is excellent, and conversions to other file formats are possible. This may also be a remedy for a major drawback of EPS: its bad storage performance. Some EPS files can be compressed to about 10% of their original size.

Open Geometry has always supported the export of 2D scenes to EPS. The new version Open Geometry 2.0 provides this possibility for 3D scenes as well. However, there are some limits to this feature.

The export from Open Geometry to EPS is very simple. Run an Open Geometry program and press <Ctrl+E> or choose the menu item "Image/Export image/Save image as EPS-file". This command works for 2D and 3D scenes. So far, the export of 2D scenes was possible with only one minor problem (transparency is lost). Almost all 2D pictures in this book were produced in this way.

The export of 3D scenes is much more delicate: In general, it will produce visibility errors. The reason for this is the following: When Open Geometry displays a 3D scene, all visibility decisions are made with the help of z-buffering. In EPS, however, this is different. There, visibility relations depend on the drawing order.[8]

If you produce an EPS file from a 3D image, you must control the drawing order, either directly in Open Geometry or after the export (with the help of a graphics package or by editing the EPS file). This can, of course, be rather laborious, and we do not recommend it for complex scenes.[9]

However, for scenes with only straight lines, curves, polygons, and other simple objects, EPS export may be a good idea. We present an example of this:

[8] This is the reason why Open Geometry's 2D graphics and EPS go together much better: Visibility decisions are made in the same way.

[9] It is almost impossible to use it when parameterized surfaces are to be displayed. The EPS-file contains hundreds of small triangles with wrong visibility. Furthermore, the surface will not be smooth-shaded.

Example 4.7. Reflection on corner
It is a well-known geometric phenomenon that a ray of light being reflected on the inside of a cube corner (three pairwise orthogonal reflecting planes) is cast back in a parallel direction (Figure 4.12). This is easy to see with the help of basic vector calculus:

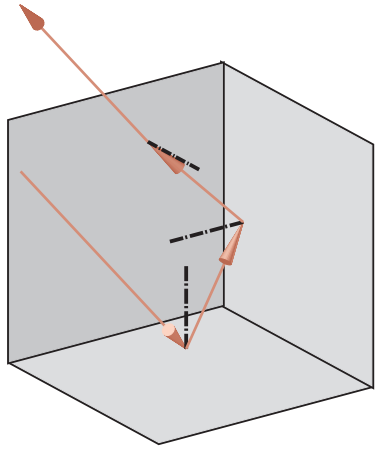

FIGURE 4.12. A ray of light is reflected on the corner of a cube and is cast back in a parallel direction.

Let the reflecting planes be the coordinate planes of a Euclidean coordinate system and $(x, y, z)^t$ be the direction of the incoming ray of light. Each reflection on a coordinate plane changes the sign of one coordinate. Thus, the reflected ray has direction $(-x, -y, -z)^t$.

This is simulated in the file `"reflect_on_corner.cpp"`. We define three global variables and initialize them in Init( ):

Listing from `"reflect_on_corner.cpp"`:

```
RegPoly3d Square[3];
StrL3d IncomingLight;
const Plane CoordPlane[3] = { XYplane, YZplane, XZplane };

void Scene::Init( )
{
    // initialize reflecting planes and incoming ray
    Real a = 4;
    StrL3d axis[3];
    axis[0].Def( P3d( a, a, 0 ), Zdir );
    axis[1].Def( P3d( 0, a, a ), Xdir );
    axis[2].Def( P3d( a, 0, a ), Ydir );
```

```
    int i;
    for ( i = 0; i < 3; i++ )
        Square[i].Def( AlmostWhite, Origin, axis[i], 4 );
    IncomingLight.Def( P3d( 5, 2, 3 ), V3d( -1, -0.5, -1.6 ) );
}
```

With respect to the later export to EPS the first thing to be done in Draw( ) is the *shading of the squares* that symbolize the cube corner planes. Only later, will we compute and draw the reflected rays:

Listing from "reflect_on_corner.cpp":

```
void Scene::Draw( )
{
    // shade reflecting planes
    int i;
    for ( i = 0; i < 3; i++ )
        Square[i].ShadeWithContour( Black, MEDIUM );

    P3d S[3];
    P3d L = IncomingLight.InBetweenPoint( -10 );
    int k;
    StrL3d lightray = IncomingLight;

    for ( k = 0; k < 4; k++ )
    {
        Real t = 1e10;
        int i0 = -1;
        for ( i = 0; i < 3; i++ )
        {
            S[i] = CoordPlane[i] * lightray;
            Real dist = lightray.GetParameter( S[i] );
            if ( dist > 1e-5 && dist < t )
            {
                t = dist;
                i0 = i;
            }
        }
        if ( i0 == -1 )
        {
            if ( k != 3 )
                Write("bad");
            else
                S[i0] = lightray.InBetweenPoint( 10 );
        }
        DrawArrowV3d( Red, L, S[i0], 1.4, 0.2 );
```

```
        if ( k < 3 )
        {
            L = S[i0];
            V3d dir = lightray.GetDir( );
            dir.Reflect( CoordPlane[i0] );
            StrL3d nrm( L, CoordPlane[i0].n( ) );
            nrm.LineDotted( Black, 0, 3, 6, ThinOrThick( 2 ) );
            lightray.Def( L, dir );
        }
    }
}
```

For the drawing of the reflected rays we compute the intersection points of the incoming rays with the reflection planes and take the point with the lowest parameter value larger than a certain small $\varepsilon > 0$. We display the corresponding ray by means of Open Geometry's ShowV3d(...) function. For better illustration we line-dot the plane normal as well.

Figure 4.12 shows the direct EPS output from Open Geometry. If you watch the image carefully, you will see that not all visibility decisions are correct: There is a small problem with the straight lines and arrows. There is no simple way of altering this. Changing the drawing order of straight line segments and pyramids would only make things worse. Of course, you could make certain changes to ShowV3d(...). For the time being, this might help, but you will always be able to find viewpoints where something goes wrong. ♢

The export to EPS uses a default PostScript line width. If you want to create an `"*.eps"`-file with very thin lines, you can change this value by editing the corresponding line in `"og.ini"` (compare Section 7.3).

POV-Ray

Open Geometry is a program for the visualization and animation of complex geometric objects. We believe that it does a good job with that. However, it has not been written for the rendering of photorealistic scenes. You have just one light source, diverse cameras, and the use of texture maps. Yet, these options are too limited for highly realistic images.

Still, in case you feel the need for some special effects with the objects you have created in Open Geometry, the new version has something in store for you. You can create an include file for the *Persistance of Vision Raytracer*, a freeware ray-tracing program usually referred to as POV-Ray. There exist versions for various platforms that can be downloaded from `http://www.povray.org/`.

The basic ideas of ray-tracing programs and Open Geometry are rather far apart. Ray-tracing aims at highly realistic images, taking into account different

materials and surface properties, different kinds of lights, shadows, reflection phenomena, and much more. The drawback is, of course, the computation time. The image is computed pixel by pixel. The rendering of a scene of reasonable size with a few transparent and/or reflecting objects takes at least some seconds. Even nowadays, the computation times for complex scenes may increase to several hours.

With hardware support, most OPEN GEOMETRY programs can be animated in absolute real time; i.e., you will get at least 20 frames per second. The resulting output is of high quality but not photorealistic. In an attempt to combine the advantages of both concepts, we implemented the possibility of exporting OPEN GEOMETRY objects to POV-Ray. You are able to study a complicated parameterized surface from all sides in OPEN GEOMETRY. If you finally decide on a good viewpoint, you can export it to POV-Ray and render a high-quality image with texture mapping, shadows, reflections, and specular points.

POV-Ray gives you all possibilities of a modern ray-tracing program. In fact, there are so many features that we can present only the most basic elements. After reading through this section, you will be able to render OPEN GEOMETRY objects in POV-Ray. However, for more detailed information, please, consult the POV-Ray documentation that comes with any regular POV-Ray distribution.

We believe that it is worthwhile having a quick glance at page 344. There, we prepared a whole page of OPEN GEOMETRY objects that have been rendered with the help of POV-Ray.

Basically, there are two steps in the creation of a POV-Ray image. First, you have to create a plain text file describing the scene in POV-Ray's *scene description language*. Then POV-Ray will read the file and compute the image. The second step — the actual rendering — is a bit different from platform to platform but not really difficult. Therefore, we will focus on step one only.

POV-Ray's scene description language is not too far away from C or C++. It is highly intuitive and easy to read. You will soon see a few examples of that. The plain text file is usually characterized by the extension `"*.pov"`. You will find a file of that type in `"DATA/INC/"`. It is called `"mask.pov"` and is derived from one of POV-Ray's ready-made scenes. We give a complete listing and a detailed discussion:

```
#version 3.1;

// Some standard include files;
// you will find more of them in
// POV-Ray's "include"-directory
#include "colors.inc"
#include "finish.inc"
#include "glass.inc"
#include "golds.inc"
```

```
#include "metals.inc"
#include "shapes.inc"
#include "stones.inc"
#include "textures.inc"
#include "woods.inc"

global_settings { assumed_gamma 1.0 }

// ----------------------------------------

camera
{
  location  <28, 12, 18>
  look_at   <0, 0, 0>
  angle 20                // fovy angle
  //orthographic          // switch to normal projection
}

light_source
{
  <400, 400, 200>
  color White
  // shadowless
}

// Blue sky sphere to avoid pure black background
sky_sphere
{
  pigment
  {
    gradient y
    color_map
    {
      [0.0 color blue 0.6]
      [1.0 color rgb <1, 1, 1>]
    }
  }
}

// ----------------------------------------

sphere
{
  <0, 0.5, 0>, 1.5      // Center and radius
  texture
  {
    pigment { color Green }
  }
}
```

```
box
{
    < −2, −1.5, −2>,   // Near lower left corner
    <  2, −1,  2>     // Far upper right corner
    texture
    {
      T_Stone25         // Predefined from stones.inc
      scale 4           // Scale texture
    }
    rotate <0, 20, 0>  // Rotate through 20 deg about y-axis
}


//#declare T_StandardSurface =
//    texture{ pigment{ color <1, 0.8, 0.2> } }
//#declare T_StandardCAD3D =
//    texture{ pigment{ color <0.5, 0.8, 1> } }
//#include "name.inc"
```

After some comments at the top, we include a couple of standard files. There, you will find a number of predefined textures and shapes. At the beginning, it is not worthwhile defining your own textures. You will find templets for almost anything you can imagine in the include files. Just read through one or the other file to see what is in store for you.

The global setting of an assumed gamma value is a more advanced feature, and you need not really care about it. It helps balancing different degrees of brightness on various platforms.

Next, we define the camera and one light source. They are initialized with Open Geometry's default values. Note, however, that POV-Ray uses a *left-handed coordinate system* with y pointing upwards (Figure 4.13). In order to transfer coordinate values from Open Geometry to POV-Ray, you have to *swap y and z*.

One light source is usually not enough to render a realistic scene. You should add more sources to cast light in the dim corners of the image. Perhaps the POV-Ray keyword shadowless will be useful. The corresponding light source will brighten up the image but cast no shadows.

In most cases it is advisable to add some kind of background object to the scene. Otherwise, everything will be embedded in pure black surroundings reminiscent of outer space but without stars.[10] In `"mask.pov"` we used a sky sphere for

[10]Of course, POV-Ray has star field textures as well, but even they have to be mapped to some kind of object.

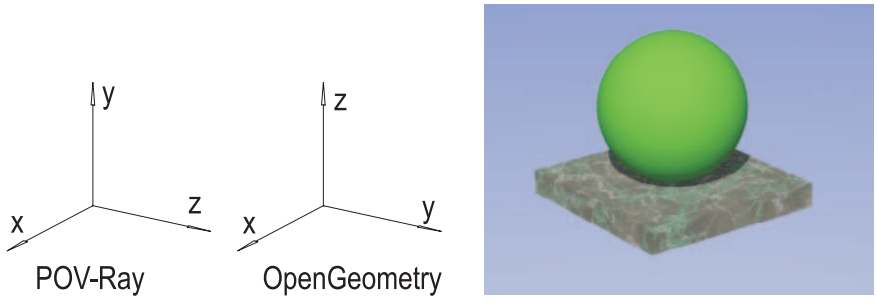

FIGURE 4.13. POV-Ray coordinate system and a simple scene (`"mask.pov"`). The sphere is of radius 1.5 and is centered at $(0, 0.5, 0)^t$. The box is defined by two opposite corners $(-2, -1.5, -2)^t$ and $(2, -1, 2)^t$ and rotated about the y-axis through 20°. Hence, it touches the sphere (all coordinates with respect to POV-Ray's coordinate system).

that purpose. The sky sphere's texture has a color map that, again, is not really absolutely POV-Ray basic.

Now we have prepared everything and can start to add some objects to the scene. It is easy to determine their geometry from the above listing: A sphere centered at $(0, 0.5, 0)^t$ of radius 1.5 and a box defined by two opposite corners. The box is rotated about the y-axis through 20°. Do not forget that all coordinates are taken with respect to POV-Ray's coordinate system! That is, the sphere lies on top of the box (Figure 4.13). The sphere's texture is very simple, since we specify only its color. For the box we use a predefined texture from `"stones.inc"`.

So far, so good. We have created a little POV-Ray scene, but we have not yet included any OPEN GEOMETRY object. In order to do this, we have to start in an OPEN GEOMETRY window. There we use the menu item "Image→Export Image→Export image as PovRay-file" or the key combination <`Ctrl+P`>. This creates a POV-Ray include file that is identified by the extension `"*.inc"`. The same effect can be achieved by calling PreparePovRayFile(...). This may be useful for animated scenes. For example you can insert something like

```
if (FrameNum( )==26 )
   PreparePovRayFile( "filename.inc" );
```

in Animate( ). We are planning to implement a method for the automatic creation of POV-Ray animations. As with to SaveAsBitmapAnimation(...) you will be able to create a series `"001.inc"`, `"002.inc"`...`"020.inc"` of POV-Ray include files that can be processed by POV-Ray animation tools.

You can include your `"*.inc"` file in your scene by commenting out the last line of `"mask.pov"` and inserting the correct path and name (e.g., `"D:/OPENGEOM/surface.inc"`). If your scene contains parameterized surfaces,

you have to comment out the line where the texture T_StandardSurface is defined. If you want to render CAD3D objects, you need the T_StandardCAD3D texture as well.

Your `"*.pov"`-file should now compile without any difficulties. You will see that all surfaces and all CAD3D objects have the same standard texture. Other objects like polygons and lines have the color you have specified in OPEN GEOMETRY.

So far, you have not needed to know the contents of the include files produced by OPEN GEOMETRY. If, however, you want additional effects, you will have to make some changes in them. In order to do that, you should know something about how OPEN GEOMETRY objects are described in POV-Ray's scene description language.

Straight lines (line segments, curves) are a simple thing in OPEN GEOMETRY; in POV-Ray they do not exist. The reason for this is clear: A straight line or line segment is a geometric object of dimension one and simply disappears in a ray-traced scene. Hence, OPEN GEOMETRY exports line segments as cylinders of revolution. A typical line in Pov-source reads

```
cylinder {
    <−7.0451914486, −2.3935121970, 0.0000000000>,
    <7.0451914486, 7.5698930991, 0.0000000000>,
    0.10
    pigment { color rgbt <0.00, 0.00, 0.00, 0.00 > }
}
```

(We inserted a few line breaks for better readability.) It defines a cylinder of revolution with the specified points as center of base and top circle and with radius 0.10. The cylinder's color is black, and it is not transparent. This is specified by the rgbt color vector $< 0.00, 0.00, 0.00, 0.00 >$. The first three entries give the RGB value of the color; the last entry $t = 0.00$ is responsible for the transparency. The relation to OPEN GEOMETRY's opacity value o is given by $t = 1 - o$; i.e., the standard opacity 1 (no transparency) corresponds to $t = 0$.

It may turn out that the line widths in the POV-Ray rendering do not look too good. This happens frequently if you use different camera positions in OPEN GEOMETRY and POV-Ray. In this case, you can change the value of Global.PovRayWidth by calling the OPEN GEOMETRY routine

ChangePovRayLineWidth(*Real* k);

The default value of Global.PovRayWidth is k = 1. It can be changed by editing `"og.ini"` (compare Section 7.3).

Geometric primitives like polygons, spheres, cylinders, and cones are exported directly to the corresponding POV-Ray objects. If you want to change their texture options, you have to identify them in the include file and edit the texture

statement. In order to help you with that, Open Geometry writes a comment line at the beginning of each new object.

By the way, ten-digit precision turned out to be necessary for the exportation to POV-Ray. Otherwise, non collinearity or coplanarity of polygonal points might be lost, which causes compile and render errors in POV-Ray. For this reason the include files are sometimes rather large, especially if your scene contains parametric surfaces or Cad3D objects.

The latter object types (parameterized surfaces, Cad3D objects) are exported to a POV-Ray *mesh object*. A mesh consists of triangles (a few hundred, a few thousand, or even more). Additionally, the normal vectors at each vertex are given in order to allow smooth-shading with soft or shiny highlights, shadows, reflections, etc. Some examples are displayed in Figure 4.14. In POV-Ray's scene description language, a mesh object generated by Open Geometry has the typical shape

```
mesh
{
    smooth_triangle
    {
        <−0.0053944707, −0.2049898854, 7.7896156440>,
        <0.9993256912, −0.0256237351, −0.0262980438>,
        <0.0000000000, −0.4207612457, 7.9889273356>,
        <0.9986159170, 0.0027662509, −0.0525223595>,
        <0.0056858564, −0.2160625444, 8.2103766880>,
        <0.9992892679, 0.0270078186, −0.0262970855>
    }
    ...
    smooth_triangle
    {
        <−0.0056858564, −0.2160625444, −8.2103766880>,
        <−0.9992892679, 0.0270078183, 0.0262970858>,
        <0.0000000000, 0.0000000000, −8.0000000000>,
        <−1.0000000000, 0.0000000000, −0.0000000006>,
        <0.0053944707, −0.2049898854, −7.7896156440>,
        <−0.9993256912, −0.0256237356, 0.0262980444>
    }
texture { T_StandardSurface }
}
```

Again, we added line breaks for better readability. If you want a different texture, you have to edit the texture statement in the last line. You can even insert something like

```
translate <5, −2, 3>
rotate <10, −30, 30>
```

before or immediately after the texture statement if you want to alter the position of the mesh.

In order to give you an overall impression, we list the complete POV-Ray file that was used for the rendering of Figure 6.18 in Chapter 6. The corresponding OPEN GEOMETRY code is `"dodecahedron.cpp"`. Note the multiple light sources that are in use to light the whole scene.

```
#version 3.1;

#include "colors.inc"
#include "finish.inc"
#include "glass.inc"
#include "golds.inc"
#include "metals.inc"
#include "shapes.inc"
#include "stones.inc"
#include "textures.inc"
#include "woods.inc"

// ------------------------------------

camera
{
  location <18.00, 10, 18>
  look_at   <0, -0.66, 0>
  rotate <0,23,0>
  angle 26
  orthographic
}

light_source
{
  <200, 360, -200>
  color Gray65
  shadowless
}
light_source
{
  <100, 100, 100>
  color Gray65
  shadowless
}
light_source
{
  <-100, 20, 0>
  color Gray40
  shadowless
}
```

```
light_source
{
  <100,200, −100>
  color Gray60
  shadowless
}
light_source
{
  <0, 130, 180>
  color Gray65
  //shadowless
}
light_source
{
  <0, 100, 200>
  color Gray65
  //shadowless
}
// Blue sky sphere to avoid pure black background
sky_sphere
{
  pigment
  {
    gradient y
    color_map
    {
      [0.0 color White ]
      [1.0 color White ]
    }
  }
}

// −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

#declare T_StandardSurface =
    texture{ pigment{ color <1, 0.6, 0.5> }  }
#declare T_StandardSurface2 =
    texture{ pigment{ color <0.7, 0.8, 1> } }
#declare T_StandardSurface3 =
    texture{ pigment{ color <1, 0.5, 0.6> } }
#declare T_StandardCAD3D =
    texture{ pigment{ color <1.0, 0.7, 0.4> }
             finish {specular 1 roughness 0.001} }
#include "../../og1_5/tmp.inc"
```

Finally, a few words on the relation between Open Geometry and POV-Ray. As far as geometric primitives are concerned, almost everything can easily be done

in POV-Ray without the help of Open Geometry. There even exist POV-Ray tools for rendering surfaces given by parametric or implicit equations.[11]

However, if you are used to working with Open Geometry, you will soon miss the possibility of changing the camera position in real time when rendering a POV-Ray image. In order to watch the opposite side of the surface, you have to change the camera coordinates and render the whole scene again. Things are even worse if you want to render a parameterized surface with counter lines. You are not allowed to change the eye point; otherwise, the counter line is just some surface polygon and looks rather weird.

Therefore, we believe that Open Geometry is a good tool for creating mesh objects of surfaces. In return, POV-Ray provides excellent rendered images of Open Geometry surfaces. Animations in POV-Ray are possible but rather cumbersome in comparison with Open Geometry. But that is an indication of the different philosophies of both programs.

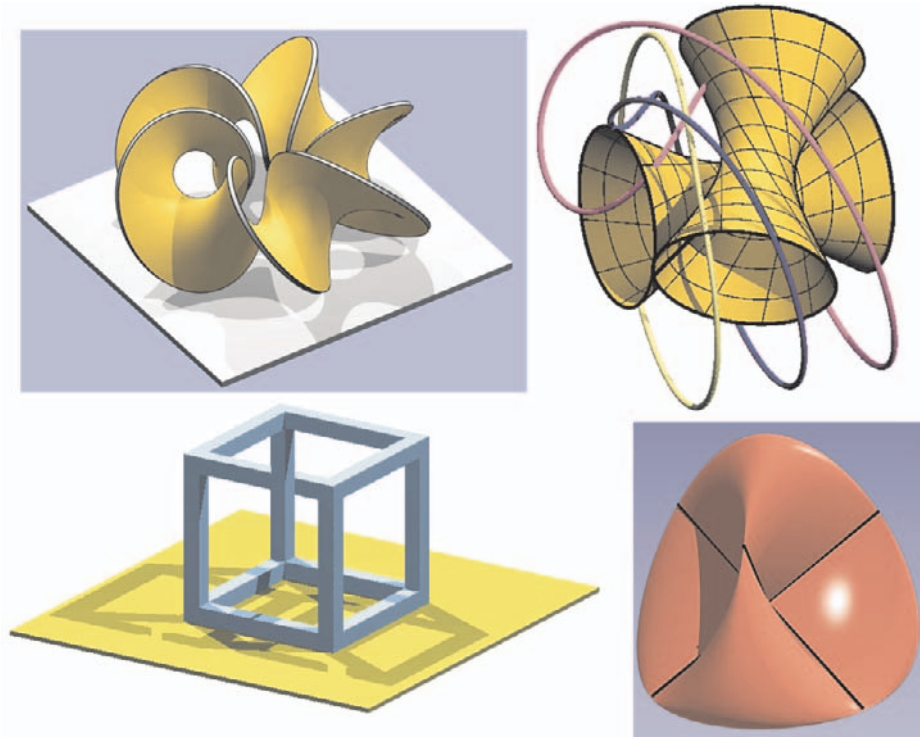

FIGURE 4.14. A few POV-Ray images of Open Geometry objects.

[11] They are, however, not included in the standard distribution.

4.3 Solids, Boolean Operations with Solids

OPEN GEOMETRY can read solid objects that were created via CAD3D. This 3D-CAD system was written by Hellmuth STACHEL, a coauthor of the original OPEN GEOMETRY book ([14]), and others, among them Georg GLAESER. The enclosed CD provides you with a DOS version of the program. A WINDOWS version exists but is not included in this package.

CAD3D is not a professional but rather an educational program,[12] and thus easy to understand. Scenes are stored in binary files with suffix `"*.llz"`. Single solids of the scene can additionally be stored (then the file has the suffix `"*.llx"`). In the directory `"DATA/"` you find two subdirectories `"DATA/LLZ/"` and `"DATA/LLX/"` with dozens of stored images. Figures 4.15 and 4.16 show only a few of them.

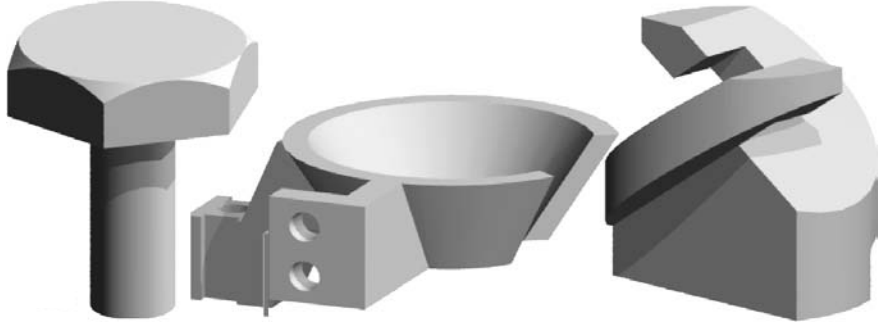

FIGURE 4.15. Some of the objects of the `"DATA/LLX/"` directory, read via OPEN GEOMETRY and exported to POV-Ray. With fast graphics hardware, all the above objects can be smooth-shaded by OPEN GEOMETRY in real time.

How to create solids directly in Open Geometry code

One of the strengths of OPEN GEOMETRY is its ability to combine properties of normally different kinds of programs. For example, there are many really good (and expensive) CAD systems on the market. And there are many good programs that allow other geometrical input. Mostly, however, there is a gap between these two directions.

Figure 4.17 illustrates how OPEN GEOMETRY fills that gap: A straight line that rotates about a skew axis generates a hyperboloid of revolution. In OPEN GEOMETRY it is easy to display a surface like a hyperboloid, even if we do not have explicit equations for it. The following listing shows how a new class Hyperboloid can be introduced:

[12] In 1993 and 1994, CAD3D won two European software prizes for educational programs.

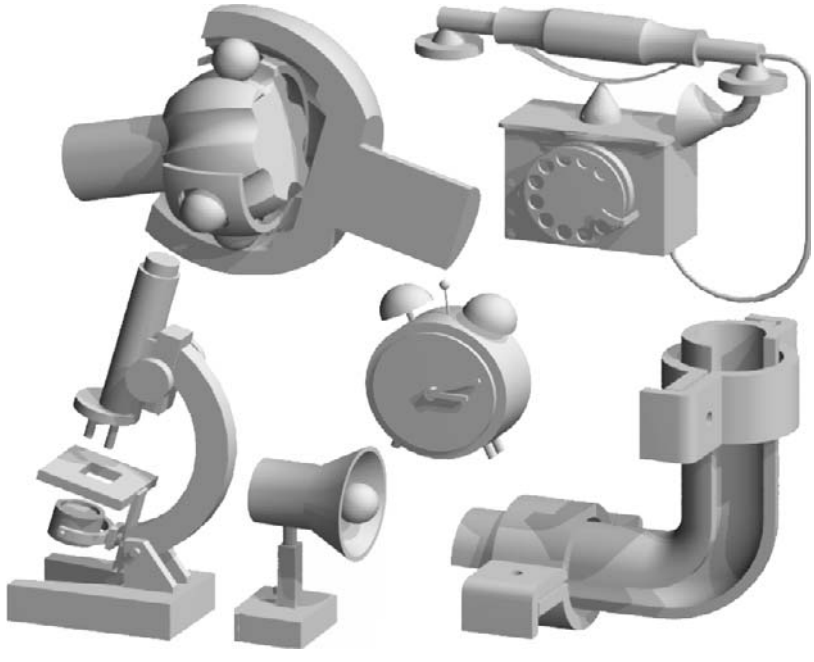

FIGURE 4.16. Some of the objects of the `"DATA/LLZ/"` directory, read via OPEN GEOMETRY and exported to POV-Ray. With fast graphics hardware, all the above objects can be smooth-shaded by OPEN GEOMETRYin real time.

Listing from `"hyperboloid.cpp"`:

```
class Hyperboloid: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real u, Real v )
   {
      // This is the kinematic generation of a hyperboloid:
      const Real a = 10, k = 1;
      P3d P( a, u, k * u ); // generating line in the plane x = a
      P.Rotate ( Zaxis, v ); // sweep function
      return P;
   }
};
```

Three quarters of the surface, enclosed by planes parallel to the base plane ($z = \pm 10$), can be defined with the following code:

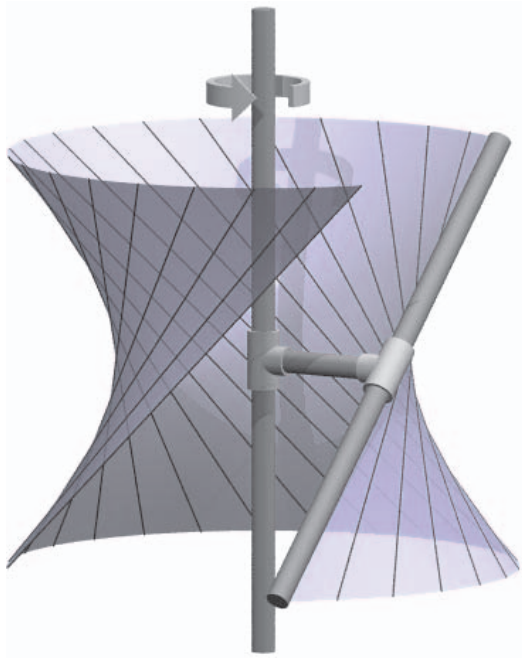

FIGURE 4.17. A hyperboloid is created by means of rotation of a straight line about a skew axis.

Listing from `"hyperboloid.cpp"`:

```
Hyperboloid Hyp;

void Scene::Init( )
{
    Hyp.Def( Yellow, 50, 100, -10 * sqrt(2), 10 * sqrt(2),
                                                90, 360 );
    ...
}
```

It can then be displayed as follows:

Listing from `"hyperboloid.cpp"`:

```
void Scene::Draw( )
{
```

```
    ...
    Hyp.Shade( SMOOTH, REFLECTING );
    Hyp.ULines( Black, 20, THIN );
    ...
}
```

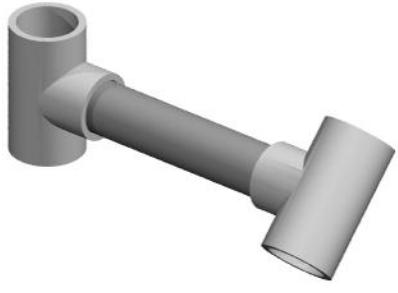

FIGURE 4.18. A detail from the preceding image.

And now the other part: You want to display a realistic mechanism that allows the rotation of a rod (Figure 4.18). Therefore, you need to define two (congruent) joints by means of a CAD system.

You have two choices: The first is to start the CAD3D system that comes with the enclosed CD, create the object there, and save it as an `"*.llz"` (or in this simple case also as an `"*.llx"`) file. In fact, we have done that all the time, since, to be honest, it is still easier to create such objects with a CAD system. The round arrow you can see in Figure 4.17 is such a solid that is read from a file.

The other choice is to create the solid "live" by means of OPEN GEOMETRY.[13] First we declare two global variables:

Listing from `"hyperboloid.cpp"`:

```
BoolResult Tube1, Tube2;
```

[13]Theoretically, OPEN GEOMETRY can create everything that CAD3D can create. In practice, however, there are numerical problems that arise due to the fact that CAD3D sometimes uses 6-byte reals instead of either 4-byte reals or 8-byte reals.

We have to keep in mind that they are pointer variables (see the compendium in Chapter 6). The following listing can, we believe, be read without much explanation, since there are comments at the critical lines. Just one thing: We first define the cylinders that form the solids via Open Geometry. Then we apply Boolean operations to these polyhedra. The result is of type *BoolResult*.

Listing from `"hyperboloid.cpp"`:

```
int order = 60;
RegPrism Cyl1, Cyl2;
Cyl1.Def( Gray, 1.1*r, 4, order );
Cyl1.Translate( 0, 0, -2 );
Cyl2.Def( Gray, r, 2, order );
Cyl2.Rotate( Yaxis, 90 );
// Calculate the union of the two cylinders
BoolResult Exterior = Cyl1 + Cyl2;
// Now shrink the cylinders
Real s = 0.8;
Cyl1.Scale( s, s, 1.1 );
Cyl2.Scale( 1.1, s, s );
// Calculate the union of the two smaller cylinders
BoolResult Interior = Cyl1 + Cyl2;
// Get the difference, i.e., the tube
Tube1 = *Exterior - *Interior;
Tube1->Rotate( Zaxis, 90 );
// Now the second solid
Tube2 = *Exterior - *Interior;
Tube2->Rotate( Yaxis, 90 );
Tube2->Rotate( Xaxis, -90 );
Tube2->Rotate( Yaxis, 90 - angle );
Tube2->Translate( 0, 10, 0 );
```

The two solids can finally be shaded as follows:

Listing from `"hyperboloid.cpp"`:

```
Tube1->Shade( );
Tube2->Shade( );
```

4.4 Texture Mapping

The basics of texture mapping with OPEN GEOMETRY have been explained in [14], p. 316. A good sample file for reference can be found on page 337 of this book. In this section, we present two more examples:

Example 4.8. How to map images onto a cylinder
We want to map two images (bitmap files) onto a cylinder, or, more precisely, onto a regular prism that approximates a cylinder of revolution. In order not to offend anyone, we took pictures of ourselves (Figure 4.19, left).

These pictures fulfill two conditions:

- Their widths and heights in pixels are a power of 2 (e.g., 128×256). This is due to OPENGL conventions (otherwise, the images are cut).
- In order to enable transparency, the background of the images was filled with a bright color with RGB values greater than 230 (e.g., with white color $= < 255, 255, 255 >$).

FIGURE 4.19. The two images on the left are mapped onto a regular prism.

Now we write an OPEN GEOMETRY-application `"map_onto_cyl.cpp"`. First, we need some global variables:

Listing from `"map_onto_cyl.cpp"`:

```
const int N = 16;
// We map two images onto 'cylinder',
// or more precisely: onto an N-sided prism
Poly3d Poly; // this poly creates the prism when it is rotated N times

TextureMap Map1, Map2; // the two images with some white background
```

The larger the order N of the prism, the slower the application will run. In the initializing part we now load the images:

Listing from "map_onto_cyl.cpp":

```
Boolean transparent = true;
unsigned char rgb_max = 230; // transparent beyond that value
Map1.LoadFromBitmapFile( "BMP/gg.bmp", transparent, rgb_max );
Map2.LoadFromBitmapFile( "BMP/hp.bmp", transparent, rgb_max );
```

When we set transparent = true, colors with RGB values greater than rgb_max are ignored (or, more precisely, are completely transparent).

Next, we define a polygon (actually a rectangle) that generates a regular N-sided prism when it is rotated N times about the z-axis through 360/N degrees.

Listing from "map_onto_cyl.cpp":

```
const Real height = 15;
Real radius = height / N;
Poly.Def( PureWhite, 4 ); // NoColor is also fine,
      // any other color will have influence on the image!
Poly[1]( radius, 0, 0 );
Poly[2]( -radius, 0, 0 );
Poly[3]( -radius, 0, height );
Poly[4]( radius, 0, height );
Real central_angle = Arc( 360. / N );
Real dist_of_side_of_the_prism = radius / tan ( 0.5 * central_angle );
Poly.Translate( 0, dist_of_side_of_the_prism, -height/2 );
```

The drawing part needs some explanation. OpenGL texture mapping in transparency mode works correctly only if the following rules are obeyed:

1. You have to draw back to front, i.e., when two textured polygons overlap, the one that is in the back has to be drawn first!
2. Transparency must be activated somehow, e.g., by means of the routine SetOpacity(0.99). There will be no visible difference to deactivated transparency mode.

In general, it can be tricky to fulfill condition 1. In our case, though, it is not very difficult: We check the position of the polygon's barycenter B and determine its distance from a plane that is perpendicular to the main projection ray (in the program it is called mid). When B lies to the rear of this plane, the whole face is a backface.

In a first loop we plot all backfaces; in a second one, only the frontfaces.

The rest is done with the polygon's method ShadeWithTexture(...), which takes quite a few parameters. For better understanding, the parameters have readable names. In principle, we drag a rectangular clipping window over the image(s). Just skim over the lines until you see what is done in detail:

Listing from `"map_onto_cyl.cpp"`:

```
V2d translate( 0, 0 );
const V2d scale( 4.0 / N, 1 );
const Real angle = 0;
const Boolean repeat = true;
SetOpacity( 0.999 ); // necessary for transparency
int i, k;
V3d n = TheCamera.GetProjDir( );
n.z = 0;
Plane mid( Origin, n ); // With the help of this plane, we
   // can judge whether a polygon is 'on the back side' or not
for ( k = 0; k < 2; k++ )
{
   // k = 0: plot only the backfaces of the cylinder
   // k = 1: plot only the frontfaces
   for ( i = 0; i < N; i++ )
   {
      P3d B;
      B = Poly.GetBaryCenter( );
      if ( ( k == 0 && B.DistFromPlane( mid ) < 0 )
            || ( k == 1 && B.DistFromPlane( mid ) > 0 ) )
      {
         Boolean first_map =
            ( i < N / 4 || i >= N / 2 && i < ( 3*N ) / 4 );
         Poly.ShadeWithTexture( first_map ? Map1 : Map2,
                        scale.x, scale.y, angle,
                        translate.x,  translate.y, repeat );
      }
      translate.x += scale.x;
      if ( translate.x >= 0.99 )
         translate.x = 0;
      Poly.Rotate( Zaxis, 360. / N );
         // after N rotations, the polygon
         // is again in the original position
   }
}
```

♢

Example 4.9. Tread
In `"tread.cpp"` we present a combination of both texture mapping and 3D animation. We map a texture on a tread running around two rollers. The rollers are cylinders of revolution implemented as objects of type *RegPrism*:

Listing from `"tread.cpp"`:

```
RegPrism Roll1, Roll2;
const Real R1 = 4.0, R2 = 2.8;
const int Switch = ( R1 > R2 ? 1 : -1 );
const Real Z1 = 5.0, Z2 = -3.0;
const StrL3d Axis1( P3d( 0, 0, Z1 ), Ydir );
const StrL3d Axis2( P3d( 0, 0, Z2 ), Ydir );
```

Their radii, their axes, and the axes' z-coordinates are global constants. Switch helps us to differentiate the cases R1 > R2 and R2 > R1. The cylinders' height or Width as we shall rather call it (their axes are parallel to y) will be computed in Init( ) for a special reason: We intend to use a bitmap of size 128×256 (OpenGL standards require bitmaps with powers of 2 as side lengths). After having computed the total length of the tread in Init( ), we adjust Width to ensure that the whole bitmap is mapped on the tread exactly $n =$ ImageNumber (global constant!) times. The heart of the program is the following function:

Listing from `"tread.cpp"`:

```
P3d PointOnTread( Real u, Real v )
{
  if ( U0 <= u && u < U1 )
    return B1 + u * V1 + v * Ydir;
  if ( U1 <= u && u < U2 )
  {
    u = ( u - U1 - Switch * R1 * Alpha ) / R1;
    return  P3d( -R1 * cos( u ), v, Z1 + R1 * sin( u ) );
  }
  if ( U2 <= u && u < U3 )
    return A2 + ( u - U2 ) * V2 + v * Ydir;
  if ( U3 <= u && u <= U4 )
  {
    u = ( u - U3 + Switch * R2 * Alpha ) / R2;
    return P3d( R2 * cos( u ), v, Z2 - R2 * sin( u ) );
  }
  else
  {
```

```
            while ( u > U4 )
                u -= U4;
            while ( u < 0 )
                u += U4;
            return PointOnTread( u, v );
        }
    }
```

If you develop the tread into a plane, you will get a rectangle R. We connect a Cartesian coordinate system with R in a way that the x-axis coincides with one of the smaller sides and the y-axis with the middle line parallel to the longer sides. Then PointOnTread(...) returns the point on the tread that corresponds to the point with coordinates (u, v). The function uses a number of global variables, and in order to understand their meaning and the ideas behind PointOnTread(...), we need to have a closer look at 2D geometry.

Obviously, the following problem is crucial to our program: Given two circles c_1 and c_2, determine their common tangents! There exist four solutions to the problem, but not all of them need be real. Furthermore, at least two of them have no relevance to our purposes. In Figure 4.20 you can see a geometric construction of the two solution tangents t_1 and t_2. We introduce an auxiliary circle c^* of radius $r_2 - r_1$. Then we determine the tangents of c^* through the center M_2 of c_2. They are parallel to the solution tangents t_1 and t_2. The points of tangency lie on the corresponding diameters of c_1 and c_2.

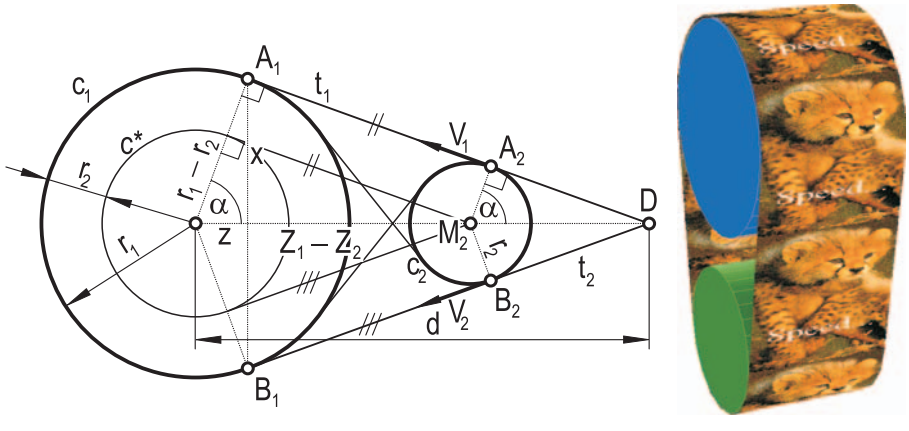

FIGURE 4.20. Constructing the common tangents of two circles and using this for a texture map plus animation.

From this drawing we get all the information we need for the computation of the common tangents in Init( ):

Listing from `"tread.cpp"`:

```
if ( R1 < 0 || R2 < 0 )
   SafeExit( "radius negative!" );

const Real dist = fabs( Z1 - Z2 );

if ( dist < R1 + R2 )
   SafeExit( "rolls too close!" );

if ( fabs( R1 - R2 ) > 0.001 )
{
   Real d = dist * R1 / ( R1 - R2 );
   Real z = R1 * R1 / ( 2 * d );
   Real x = Sqrt( R1 * R1 - z * z );
   Alpha = fabs( atan( z / x ) );
   A1.Def( -x, 0, Z1 - z );
   B1.Def(  x, 0, Z1 - z );

   d -= dist;
   z = R2 * R2 / ( 2 * d );
   x = Sqrt( R2 * R2 - z * z );
   A2.Def( -x, 0, Z2 - z );
   B2.Def(  x, 0, Z2 - z );
}
else
{
   Alpha = 0;
   A1.Def( -R1, 0, Z1 );
   B1.Def(  R1, 0, Z1 );
   A2.Def( -R2, 0, Z2 );
   B2.Def(  R2, 0, Z2 );
}

V1 = A1 - A2;
V1.Normalize( );
V2 = B2 - B1;
V2.Normalize( );
```

First, we compute the intersection point D of the tangents by means of the relation R1 − R2 : Z1 − Z2 = R1 : d (Figure 4.20). Then it is easy to determine the reals x and z that lead to the points of tangency A1, B1 on c_1 and A2, B2 on c_2. In addition, we compute the value of the angle α as we need it in PointOnTread(...). Note that we do this only if the radii R1 and R2 are different. If they are (almost) equal, we take the obvious values $\alpha = 0$, $Ai = (-Ri, 0, Zi)^t$ and $Bi = (Ri, 0, Zi)^t$. Finally, V1 and V2 are the normalized vectors in the direction of A1A2 and B1B2, respectively. Note that we catch the exceptional case without real solutions at the very beginning. In fact, we even avoid the case of intersecting rollers.

Now we have to compute the values U0, ..., U4 that are used in PointOnTread(...). They give the length between successive transition points of the tread. For example, U2 − U1 is the arc length of c_1 between U1 and U1. We need the variable Switch for that.

Listing from `"tread.cpp"`:

```
const Real length = A1.Distance( A2 );
U0 = 0;
U1 = length;
U2 = U1 + R1 * ( PI + 2 * Switch * Alpha );
U3 = U2 + length;
U4 = U3 + R2 * ( PI - 2 * Switch * Alpha );

Width = U4 / ( 2 * ImageNumber );
```

The width of our tread is computed with respect to the total length U4 of the tread. Of course, this formula works only for bitmaps with ratio height : width = 2 : 1. Finally, we initialize the rollers themselves:

Listing from `"tread.cpp"`:

```
Real r1 = 0.97 * R1;
Real r2 = 0.97 * R2;
Roll1.Def( Blue, r1, 2 * Width, 30, SOLID, Yaxis );
Roll2.Def( Green, r2, 2 * Width, 30, SOLID, Yaxis );
Roll1.Translate( 0, -Width, Z1 );
Roll2.Translate( 0, -Width, Z2 );
```

To avoid visibility problems, you may have to use a scaling factor different from 0.97. It depends on the rollers' curvature and on the approximation quality of the tread you use. This quality is determined by a global integer N. In Draw( ) we map the textures on the tread:

Listing from `"tread.cpp"`:

```
int i;
Real u, delta = U4 / N;
Real scale = delta / ( 2 * Width );
Poly3d Poly;
Poly.Def( PureWhite, 4 );
for ( i = 0, u = U; i < N; i++, u += delta )
{
   Poly[1] = PointOnTread( u,  Width );
   Poly[2] = PointOnTread( u, -Width );
   Poly[3] = PointOnTread( u + delta, -Width );
   Poly[4] = PointOnTread( u + delta,  Width );
   Poly.ShadeWithTexture( Map, 1, scale, 0, 0, i * scale, true );
}
```

Here, you have to know how Open Geometry and OpenGL handle texture mappings (see [14]). We approximate the tread by a number of small rectangles and map the corresponding part of the bitmap on them. The initial value of u in the above listing is a global variable U that is used to animate the whole thing:

Listing from `"tread.cpp"`:

```
void Scene::Animate( )
{
   U += 0.2;
   if ( U > U4 )
      U -= U4;
   Roll1.Rotate( Axis1, Deg( U / R1 ) );
   Roll2.Rotate( Axis2, Deg( U / R2 ) );
}
```

Because of the texture mapping, the animation is rather slow. You had better avoid large bitmaps. With our current hardware we get about 7 frames per second with a bitmap of size 128×256 (Figure 4.20). ♢

4.5 Advanced Animations

In this section we present a few examples that demonstrate Open Geometry's ability to simulate even very complex 3D animations. Sometimes, the difference between 3D animations and 3D kinematics is not easy to tell. Thus, you will find examples of interest in Section 4.1 as well.

Example 4.10. A developable Möbius band
In 1865 the German geometer F.A. MÖBIUS described a simple method to produce a model of a nonorientable surface ([24]). He proposed to join the opposite corners of a long, thin slip of paper (Figure 4.21). The resulting object has only one side and one edge.

It is not difficult to find mathematical surfaces that are topologically equivalent to MÖBIUS's example (`"moebius1.cpp"`). Consider an axis z, a plane ε through z, and a circle $c \subset \varepsilon$ such that the center C of c does not lie on z. Now rotate c with constant angular velocity ω about z, and a diameter D_1D_2 of c about the circle axis with angular velocity $\omega/2$. The diameter will sweep a one-sided and nonorientable band (Figure 4.21). By the way, this band is part of a right rotoid helicoid (compare [14], p.303).

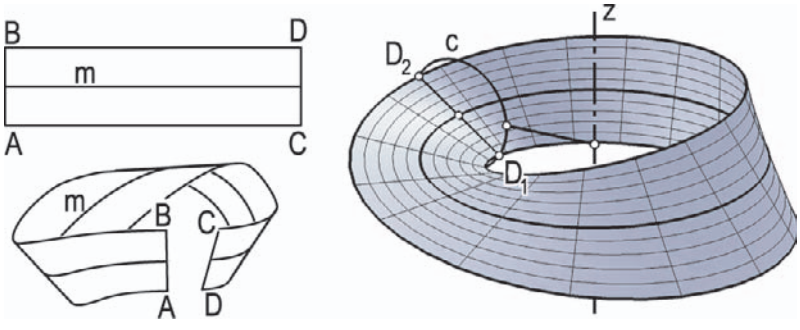

FIGURE 4.21. MÖBIUS's idea of producing a one-sided surface and a topologically equivalent right rotoid helicoid.

The program `"moebius1.cpp"` is not of so much interest to us at this point. It is just some solid OPEN GEOMETRY code. We will rather focus on certain geometric properties of the original Möbius band. Obviously, it is a torse: Its development into the plane yields the original rectangle. This is an essential difference from the example of `"moebius1.cpp"`. You can easily verify this by having a look at the contour outline in Figure 4.21. It does not consist of straight line segments. Most of the numerous examples of Möbius bands that you find in books or on the Internet do not pass this test, i.e., they are *not developable*.

In fact, it took quite a while before the existence of a developable Möbius band was proved in a strict mathematical sense. A relatively simple example was given by M. SADOWSKY in [32]. It consists of three planar parts and three parts of cylinders of revolution. However, this surface is not analytic.

The first example of a *developable* and *analytic* Möbius band was given by W. WUNDERLICH in 1962 ([38]). His example has another advantage in comparison with others: It approximates the stable position the paper band will naturally take if not subjected to exterior influence and if the material is stiff enough. This position is well defined by certain physical conditions but could

never be given in explicit form ([31]). W. WUNDERLICH's approach will allow us to visualize an approximation of this stable position. We will denote it by M. This is the only attempt to visualize the real-world Möbius band on the computer screen we know of.

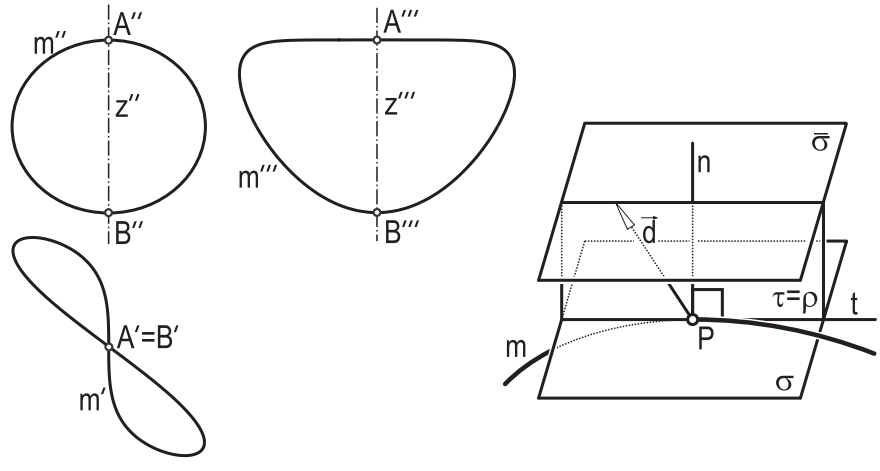

FIGURE 4.22. The midline of the Möbius band and the rectifying plane in a curve point.

WUNDERLICH starts with a close inspection of the *midline* m of the band (Figure 4.21, Figure 4.22). From several photos of a model he deduces the existence of an axis of symmetry z that intersects the band in two points A and B. In B, the axis z is perpendicular to the tangent plane of the band. Furthermore, z is the torse generator through A. Taking into account this behavior and a few other measurements, he proposes a rational parametric representation for the approximation of m. Using homogeneous coordinates $x_0 : x_1 : x_2 : x_3 = 1 : x : y : z$ it can be written as

$$\begin{aligned} x_0 &= \tfrac{1}{2}\left(1 + d^2u^2 + 2deu^4 + e^2u^6\right), \\ x_1 &= au + bu^3 + cu^5, \\ x_2 &= du + eu^3, \\ x_3 &= -C, \end{aligned}$$

where the coefficients a, b, c, d, e, and C have the values

$$a = \tfrac{1}{2}, \quad b = \tfrac{1}{3}, \quad c = \tfrac{1}{6}, \quad d = \tfrac{2}{3}, \quad e = \tfrac{1}{3}, \quad \text{and} \quad C = \tfrac{4}{5}. \tag{3}$$

The parameter u ranges in $\mathbb{R} \cup \{\infty\}$, and the resulting curve m is closed and rational of order six. Its top, front, and side views are displayed in Figure 4.22.

For the next step it is important to note that m is a *geodetic line* on the band. This follows immediately from the fact that the development of m yields a straight

line. Elementary differential geometry tells us that the osculating plane $\sigma = \sigma(u)$ of m is perpendicular to the tangent plane $\tau = \tau(u)$ of M. Not only is τ the tangent plane of M, but the *rectifying plane* $\varrho = \varrho(u)$ of m as well (Figure 4.23).[14] In other words; M is the *rectifying torse* of m (the envelope of the rectifying planes of m).

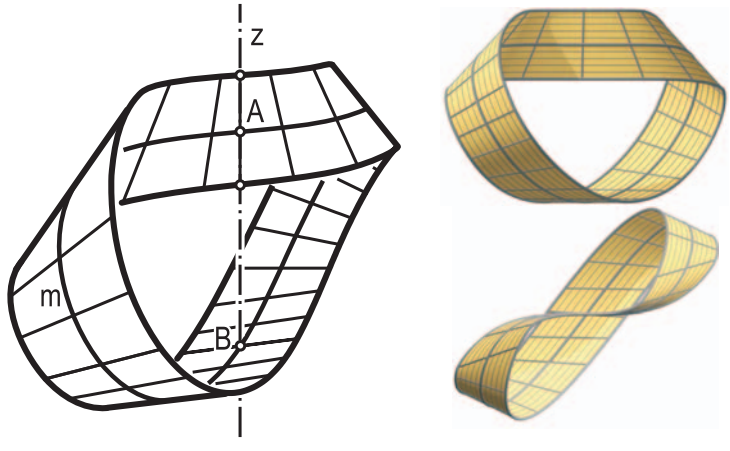

FIGURE 4.23. WUNDERLICH's developable and analytic model of the stable position of the Möbius band.

From these considerations one can conclude that M is a torse of class 21 and order 39. It is, of course, useless to compute a parametric representation of M. But with OPEN GEOMETRY this is not necessary at all. We have already enough information to write a program (`"moebius2.cpp"`). We declare the coefficients (3) as global variables and implement the midline m of the band:

Listing from `"moebius2.cpp"`:

```
class MyMidLine1: public ParamCurve3d
{
public:
  P3d CurvePoint( Real u )
  {
    const Real u3 = u * u * u;
    Real x0 = 0.5 * ( 1 + d * d * u * u +
          2 * d * e * u3 * u + e * e * u3 * u3 );
    Real x1 = a * u + b * u3 + c * u3 * u * u;
    Real x2 = d * u + e * u3;
```

[14]The rectifying plane of a space curve is the plane spanned by the curve tangent and normal of the osculating plane.

```
        Real x3 = -C;
        return P3d( x1 / x0, x2 / x0, x3 / x0 );
    }
    StrL3d Tangent( Real u )
    {
        const Real u2 = u * u;
        Real d0 = u * ( 3 * e * e * u2 * u2 +
                4 * d * e * u2 + d * d );
        Real d1 = 5 * c * u2 * u2 + 3 * b * u2 + a;
        Real d2 = 3 * e * u2 + d;

        return StrL3d( CurvePoint( u ),
            P3d( d1 / d0, d2 / d0, 0 ) );
    }
    Plane OsculatingPlane( Real u )
    {
        const Real u2 = u * u;
        Real dd0 = 15 * e * e * u2 * u2 +
                12 * d * e * u2 + d * d;
        Real dd1 = 20 * c * u2 * u + 6 * b * u;
        Real dd2 = 6 * e * u;
        return Plane( P3d( dd1 / dd0, dd2 / dd0, 0 ),
            Tangent( u ) );
    }
    Plane RectifyingPlane( Real u )
    {
        Plane p = OsculatingPlane( u );
        p.Rotate( Tangent( u ), 90 );
        return p;
    }
};
MyMidLine1 MidLine1;
```

We will need the curve's tangent t, the osculating plane σ, and the rectifying plane ϱ. Usually, one would compute them by connecting "neighboring" curve points, but here we can do better. It is easy to compute exact representations for t and σ from the parameterized equation of m. The plane ϱ is obtained by rotating σ about t through 90°. Later we will have to intersect two "neighboring" rectifying planes. So it seems to be a good idea to gain some additional accuracy at this point.

Of course, we cannot exhaust the whole parameter range $\mathbb{R} \cup \{\infty\}$ of the midline m. Therefore, we need a second, parametrically transformed instance of the class *ParamCurve3d*. The transformation $u \mapsto \frac{1}{u}$ does a good job:

Listing from `"moebius2.cpp"`:

```
class MyMidLine2: public ParamCurve3d
{
public:
   P3d CurvePoint( Real u )
   {
      return MidLine1.CurvePoint( 1 / u );
   }
   StrL3d Tangent( Real u )
   {
      return MidLine1.Tangent( 1 / u );
   }
   Plane OsculatingPlane( Real u )
   {
      return MidLine1.OsculatingPlane( 1 / u );
   }
   Plane RectifyingPlane( Real u )
   {
      return MidLine1.RectifyingPlane( 1 / u );
   }
};
MyMidLine2 MidLine2;
```

Now we define both parts of the curve in Init( ). If we use the parameter interval $[-1, 1]$ twice, the whole curve will be displayed.

Listing from `"moebius2.cpp"`:

```
MidLine1.Def( Black, 100, -1, 1 );
MidLine2.Def( Black, 100, -1, 1 );
```

For the implementation of M we use the OPEN GEOMETRY class *RuledSurface*:[15]

[15] An analogous surface is defined with the help of MidLine2.

Listing from "moebius2.cpp":

```
class MyMoebiusBand1: public RuledSurface
{
public:
   virtual P3d DirectrixPoint( Real u )
   {
      return MidLine1.CurvePoint( u );
   }
   virtual V3d DirectionVector( Real u )
   {
      Plane o, r1, r2;
      o = MidLine1.OsculatingPlane( u );
      o.Translate( o.n( ) );
      r1 = MidLine1.RectifyingPlane( u - Eps );
      r2 = MidLine1.RectifyingPlane( u + Eps );
      P3d P;
      if ( o.SectionWithTwoOtherPlanes( r1, r2, P ) )
         return V3d( MidLine1.CurvePoint( u ), P );
      else
         return Origin; // dummy
   }
};
MyMoebiusBand1 MoebiusBand1, MoebiusBand2;
```

The directrix is, of course, the midline. The direction vector $\vec{d} = \vec{d}(u)$ is parallel to the intersection of two "neighboring" rectifying planes, and that is exactly how we compute it. However, we have to take care of the length of $\vec{d}(u)$ (we must ensure that the development of M yields a slip of constant width). Therefore, we choose its end point in the plane $\overline{\sigma}$: the parallel plane at distance one from the osculating plane (compare Figure 4.22).

Note that our construction yields an discontinuity in the parameterization of M. At the value $u_0 = 0$ we have

$$\lim_{u \to u_0^+} \vec{d}(u) = - \lim_{u \to u_0^-} \vec{d}(u).$$

This is no reason to worry. On the contrary, it is inherent to the topology of the Möbius band. In order to avoid any trouble in displaying the surface, we split it in two at the parameter value u_0:

Listing from `"moebius2.cpp"`:

```
const Real w = 0.35;
MoebiusBand1.Def( Green, 60, 21, -1, -0.001, -w, w );
MoebiusBand2.Def( Green, 60, 21, 0.001, 1, -w, w );
MoebiusBand3.Def( Green, 120, 21, -1, 1, -w, w );
```

The third part, MoebiusBand3, stems from the second part of the midline. Here, no splitting is necessary. The final output is displayed in Figure 4.23

Of course, it is now a very tempting task to write an animation of the folding and unfolding of WUNDERLICH's Möbius band. With the help of the preceding considerations, this is not too difficult.

Basically, we triangulate the surface and perform small rotations about certain triangle sides until we reach a planar object. This procedure is schematically displayed in Figure 4.24. We start with a mesh $\mathbf{T} = \{T_0, \ldots, T_n\}$ of triangles such that any two consecutive triangles T_i and T_{i+1} have a side $a_{i,i+1}$ in common. The angles between the supporting planes ε_i and ε_j of the triangles T_i and T_j will be denoted by φ_{ij}. In order to "develop" the mesh $\mathbf{T}$, we rotate T_{i+1} about $a_{i,i+1}$ through $\varphi_{i,i+1}$ into the plane ε_i. Then we repeat this step with the axis $a_{i-1,i}$ and the angle $\varphi_{i-1,i}$, etc. Finally, we get a mesh that lies completely in ε_0.

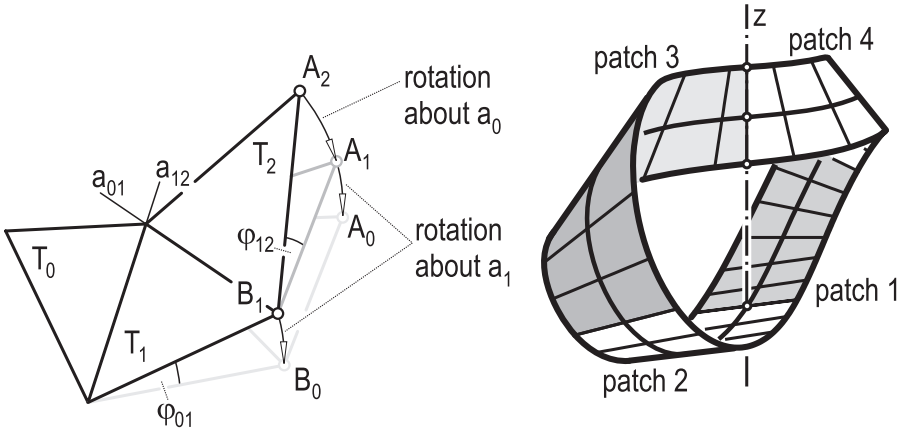

FIGURE 4.24. The basic idea of the unfolding of the Möbius band (left). We triangulate the band and rotate the triangles one by one until they are all coplanar. For the triangulation we use four patches (right). This allows optimal exploitation of the object's symmetry.

The whole idea is realized in `"moebius3.cpp"`. There, we need the parametric representation of the band as used in `"moebius2.cpp"`. However, the band

need not be implemented as a parameterized surface. Therefore, we write a list of functions that return all objects we need (curve point, tangent, osculating plane, and rectifying plane of the midline). Then we implement a separate function SurfacePoint1(*Real* u, *Real* v) according to our considerations in "moebius2.cpp".

Listing from "moebius3.cpp":

```
P3d SurfacePoint1( Real u, Real v )
{
    const Real eps = 1e-4;
    Plane o, r1, r2;
    o = OsculatingPlane( u );
    o.Translate( o.n( ) );
    r1 = RectifyingPlane( u - eps );
    r2 = RectifyingPlane( u + eps );
    P3d P;
    if ( o.SectionWithTwoOtherPlanes( r1, r2, P ) )
        return CurvePoint( u ) + v * V3d( CurvePoint( u ), P );
    else
        return Origin; // dummy
}
```

A second function SurfacePoint2(*Real* u, *Real* v) returns the point corresponding to $(1/u, v)$, and it will be used to display the second half of the band. In fact, we use four patches this time in order to overcome the difficulties with the parameterization and to make best use of the object's symmetry (Figure 4.24).

We decide on a constant integer N and allocate the memory for four arrays of triangles and two arrays of reals:

Listing from "moebius3.cpp":

```
const int N = 30;
const int N2 = 2 * N;
const int N3 = N2 - 1;

const int M = 50;

Poly3d Tri[4][N2];

Real Phi[2][N2];
```

The constant N will be one-eighth of the total number of triangles, i.e., each patch consists of N2 = 2N triangles. The constant N3 is just an abbreviation for a frequently used integer, while M determines the number of frames used for a complete animation cycle: The band will unfold completely within M frames. For reasons of symmetry we need only two patches of reals to store the angles between the supporting planes of two consecutive triangles.

Now we have to fill the arrays with the correct values. In Init( ) we start with the triangle points:

Listing from `"moebius3.cpp"`:

```
const Real w = 0.4 * Factor; // width of Moebius band
// compute triangle points
int i;
Real u;
const Real delta = ( 1.0 - 2 * 1e-4 ) / N;
for( i = 0, u = 1e-4; i < N; i++, u += delta )
{
   Tri[0][2*i].Def( LightYellow, 3 );
   Tri[0][2*i][1] = SurfacePoint1( u, w );
   Tri[0][2*i][2] = SurfacePoint1( u + delta, w );
   Tri[0][2*i][3] = SurfacePoint1( u, -w );

   Tri[0][2*i+1].Def( LightYellow, 3 );
   Tri[0][2*i+1][3] = SurfacePoint1( u + delta, -w );
   Tri[0][2*i+1][2] = Tri[0][2*i][2];
   Tri[0][2*i+1][1] = Tri[0][2*i][3];

   Tri[2][2*i].Def( LightYellow, 3 );
   Tri[2][2*i][1] = SurfacePoint2( 1 - u, w );
   Tri[2][2*i][2] = SurfacePoint2( 1 - u - delta, w );
   Tri[2][2*i][3] = SurfacePoint2( 1 - u, -w );

   Tri[2][2*i+1].Def( LightYellow, 3 );
   Tri[2][2*i+1][3] = SurfacePoint2( 1 - u - delta, -w );
   Tri[2][2*i+1][2] = Tri[2][2*i][2];
   Tri[2][2*i+1][1] = Tri[2][2*i][3];
}
for ( i = 0; i < N2; i++ )
{
   Tri[1][i].Def( LightYellow, 3 );
   Tri[3][i].Def( LightYellow, 3 );
}
SetRemainingTriangles( );
```

The triangulation of the first and third patches is computed by means of SurfacePoint1(*Real* u, *Real* v). For the remaining triangles we employ the useful function SetRemainingTriangles( ). It is implemented globally:

Listing from "moebius3.cpp":

```
P3d MyRotate( const P3d &P )
{
  return P3d( -P.x, -P.y, P.z );
}
void SetRemainingTriangles( )
{
  int k;
  for ( k = 0; k < N2; k++ )
  {
    Tri[1][k][1] = MyRotate( Tri[0][k][1] );
    Tri[1][k][2] = MyRotate( Tri[0][k][2] );
    Tri[1][k][3] = MyRotate( Tri[0][k][3] );

    Tri[3][k][1] = MyRotate( Tri[2][k][1] );
    Tri[3][k][2] = MyRotate( Tri[2][k][2] );
    Tri[3][k][3] = MyRotate( Tri[2][k][3] );
  }
}
```

It does nothing but rotate the triangles of the first and third patches about z through 180° but without using trigonometric functions. (This is very fast!) In Animate( ) we will recalculate the triangulation again and again. Since the computation of a surface point is quite laborious (we have to intersect two rectifying planes of the midline!), it is sensible to save some computing time by using SetRemainingTriangles( ).

The next step is the computation of the correct angles. It has to be done only once, in Init( ):

Listing from "moebius3.cpp":

```
V3d n1, n2;
for ( i = 1; i < N2; i++ )
{
  n1 = Tri[0][i-1].GetNormalizedNormal( );
  n2 = Tri[0][i].GetNormalizedNormal( );
  Phi[0][i] = Deg( n1.Angle( n2, true, false ) );
  CheckAngle( 0, i );

  n1 = Tri[2][i-1].GetNormalizedNormal( );
```

```
    n2 = Tri[2][i].GetNormalizedNormal( );
    Phi[1][i] = Deg( n1.Angle( n2, true, false ) );
    CheckAngle( 1, i );

    Phi[0][i] /= M;
    Phi[1][i] /= M;
  }

  // two exceptional angles
  n2 = Tri[0][0].GetNormalizedNormal( );
  Phi[0][0] = Deg( Zdir.Angle( n2, true, false ) );
  Phi[0][0] /= M;

  n1 = Tri[0][N3].GetNormalizedNormal( );
  n2 = Tri[2][0].GetNormalizedNormal( );
  Phi[1][0] = Deg( n1.Angle( n2, true, false ) );
  Phi[1][0] /= M;
```

We determine the normalized normal vectors of the supporting planes and compute their angles. In order to be sure that we have obtained the correct angle we check it in CheckAngle(0, i) and correct it if necessary. CheckAngle(0, i) performs a test rotation and checks whether the new angle is small enough. If this is not the case, the angle is multiplied by -1.

Listing from `"moebius3.cpp"`:

```
void CheckAngle( int j, int i )
{
    Poly3d poly;
    poly.Def( Black, 3 );
    poly = Tri[2*j][i];
    StrL3d axis;
    axis.Def( Tri[2*j][i-1][2], Tri[2*j][i-1][3] );
    poly.Rotate( axis, Phi[j][i] );
    V3d n1 = Tri[2*j][i-1].GetNormalizedNormal( );
    V3d n2 = poly.GetNormalizedNormal( );
    if( fabs( Deg( n1.Angle( n2, true, false ) ) > 0.01 ) )
        Phi[j][i] *= -1;
}
```

Of course, this is rather cumbersome, but it has to be done only once during the precalculations. Note that we divide the angle values immediately by M after the angle check. In each frame we will perform one rotation through the small increment angles, which results in a completely planar mesh after M frames.

Draw( ) is very simple. We shade only the triangles and draw those sides that form the borderline of the band in thin black.

Listing from "moebius3.cpp":

```
void Scene::Draw( )
{
   int i, j;
   for ( i = 0; i < N2; i++ )
   {
      for ( j = 0; j < 4; j++ )
      {
         Tri[j][i].Shade( );
         Tri[j][i].Shade( );
         Tri[j][i].Shade( );
         Tri[j][i].Shade( );

         StraightLine3d( Black, Tri[j][i][1],
            Tri[j][i][(i%2)+2], THIN );
      }
   }
   StraightLine3d( Black, Tri[2][N3][2], Tri[2][N3][3], THIN );
   StraightLine3d( Black, Tri[3][N3][2], Tri[3][N3][3], THIN );
}
```

Animate( ) needs more considerations than Draw( ). We use two global variables to control the folding and undfolding of the Möbius band:

```
int Count;
Boolean Pause;
```

They are set to 0 and **false**, respectively, when we enter Animate( ) for the first time. In order to give a good overview, we display the whole part at once:

Listing from "moebius3.cpp":

```
void Scene::Animate( )
{
   if ( !Pause )
   {
      StrL3d axis;

      axis.Def( Tri[0][0][1], Tri[0][0][3] );
      int i;
      for ( i = 0; i < N2; i++ )
      {
         Tri[0][i].Rotate( axis, Phi[0][0] );
         Tri[2][i].Rotate( axis, Phi[0][0] );
```

```
        }

        for ( i = 1; i < N3; i++ )
        {
            axis.Def( Tri[0][i-1][2], Tri[0][i-1][3] );
            int j;
            for( j = i; j < N2; j++ )
                Tri[0][j].Rotate( axis, Phi[0][i] );
            for( j = 0; j < N2; j++ )
                Tri[2][j].Rotate( axis, Phi[0][i] );
        }

        axis.Def( Tri[0][N3][2], Tri[0][N3][3] );
        for ( i = 0; i < N2; i++ )
            Tri[2][i].Rotate( axis, Phi[1][0] );

        for ( i = 1; i < N3; i++ )
        {
            axis.Def( Tri[2][i-1][2], Tri[2][i-1][3] );
            int j;
            for ( j = i; j < N2; j++ )
                Tri[2][j].Rotate( axis, Phi[1][i] );
        }
        SetRemainingTriangles( );
    }

    Count++;

    if ( Count == M ) // full animation period complete
    {
        Pause = true; // wait a little
        int i;
        for ( i = 0; i < N2; i++ ) // reverse sense of animation
        {
            Phi[0][i] *= -1;
            Phi[1][i] *= -1;
        }
    }
    if ( Pause ) // check whether break was already
                 // long enough (50 frames)
    {
        if ( Count == M + 50 )
        {
            Count = 0;
            Pause = false;
        }
    }
}
```

We have four parts where we rotate triangles. The first and third parts (those parts with only one count variable i) consist of rotations about one axis only. They occur at the transition between the different patches and have to be treated as special cases, since they don't fit exactly into the scheme of the other rotations. We rotate only the first and third patches and set the remaining triangle point "by hand."

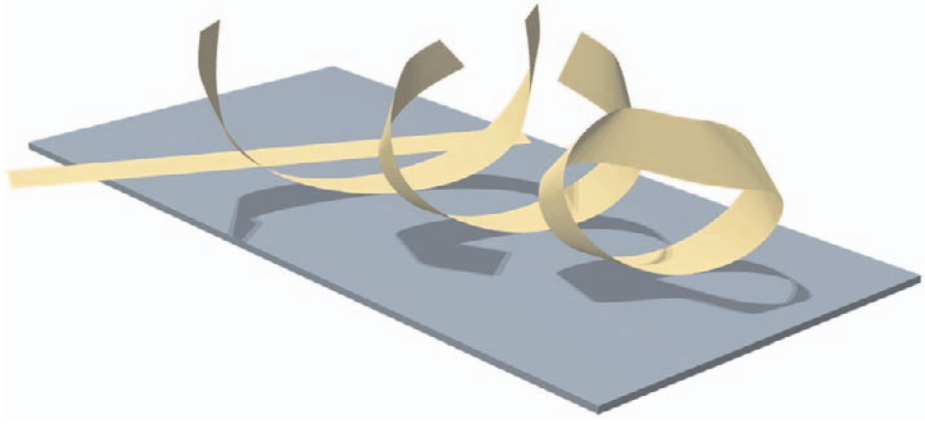

FIGURE 4.25. The folding of the stable form of the original Möbius band.

All of this happens only if the value of Pause is false. Count is increased by one in each new frame. If Count == M, the band will be completely flat. We set Pause to true and multiply all angles by -1. After a short delay of 50 frames everything starts again in the reverse direction. In Figure 4.25 we display a POV-Ray rendering of the different stages of the animation. ♢

Example 4.11. Caravan of arrows
We have already mentioned that MÖBIUS developed his band as a simple example of a one-sided surface. By means of a paper model this can easily be verified. On the computer screen, however, you still need some imagination to see this.

The program `"moebius_with_arrows.cpp"` is aimed to help you with that. Additionally, it is an example of the use of the OPEN GEOMETRY class *Arrow3d*. We display the band as in Example 4.10 and add an animation of arrows traveling along the surface. Essentially, we use the methods of `"moebius3.cpp"` to display the band. The only (minor) difference is that we introduce a *global* constant ScaleFactor for scaling the surface to an appropriate OPEN GEOMETRY size (in `"moebius3.cpp"`, this factor is local).

In addition, we must be able to bring an arrow into a certain position with respect to the Möbius band. We use the following routine to solve that task:

Listing from "moebius_with_arrows.cpp":

```
void SetLocalSystem( Real u, int state, P3d &P, V3d &x, V3d &z,
    Real a, Real b )
{
    if ( u < 0 )
    {
        Real t = u + 1;
        P = MidLine1.CurvePoint( t );
        x = MidLine1.Tangent( t ).GetDir( );
        z = MidLine1.RectifyingPlane( t ).n( );
    }
    else
    {
        Real t = 1 - u;
        P = MidLine2.CurvePoint( t );
        x = MidLine2.Tangent( t ).GetDir( );
        z = MidLine2.RectifyingPlane( t ).n( );
    }
    if ( fabs( u ) <= 1 ) // correct orientation of the tangent
        x *= -1;
    V3d y = x ^ z;
    z *= b * state;
    if ( u > 1 )
        z *= -1;
    P *= ScaleFactor;
    P.Translate( a * y );
    P.Translate( z );
};
```

SetLocalSystem(...) defines a point P and two vectors x, z that can be used to determine the position of an arrow according to Figure 4.26. We have to take into account the splitting of the band (compare Example 4.10) and the orientation of the curve and curve tangent.

The input parameters a and b determine the distance from the arrow tip to the rectifying plane ϱ of the midline and from the tangent plane τ of the band to the corresponding curve point Q. The vector x is parallel to the midline tangent; z is parallel to the surface normal. The meaning of the input parameters u and state will become clear after a closer look at the animation part.

Before we can describe what we are doing, we have to describe our intention. This can best be done by referring the reader to Figure 4.27. You can see a Möbius band with two pairs of arrows and one with a long "caravan" of arrows. In "moebius_with_arrows.cpp" the arrows move around the band, and you can switch between the two different types. For the animation we use the following global constants and variables:

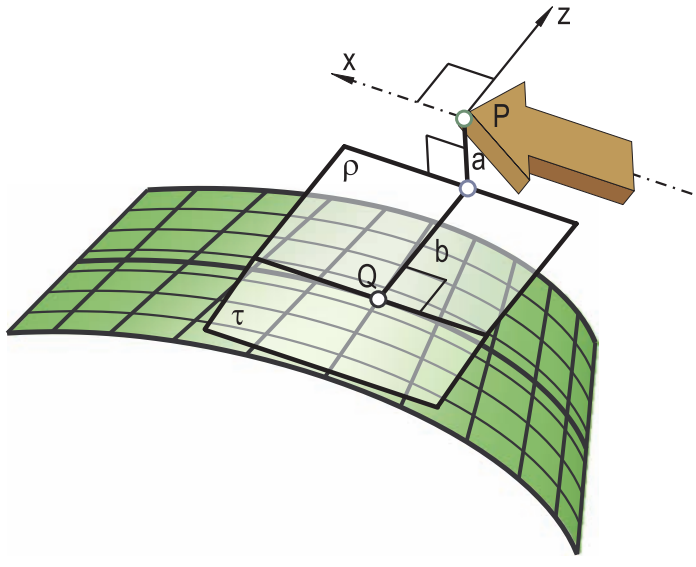

FIGURE 4.26. Assigning an arrow to the Möbius band.

Listing from `"moebius_with_arrows.cpp"`:

```
const int N = 18;

const Real U1 = -2;
const Real Delta = (Real) ( - 2 * U1 ) / ( N - 1 );
Real Umax;
Boolean ShowCaravan;

Real U;
int State;
```

The constant N is half the number of arrows in the caravan case. The animation parameter U will vary either in $[\mathsf{U1}, \mathsf{Umax}]$ where $\mathsf{Umax} = -\mathsf{U1}$ in the *pair mode* or where $\mathsf{Umax} = \mathsf{U1} + \mathsf{Delta}$ in the *caravan mode*. The integer State takes the values ± 1. In pair mode, it is used to determine the current side of an arrow.

We globally declare an instance Arrow of class *Arrow3d* and define it in Init( ):

Listing from "moebius_with_arrows.cpp":

```
Arrow.Def( 1 );
Arrow.Scale( 2 );
```

The integer argument in Arrow.Def(1) determines the arrow type. You have a choice among four different shapes (compare Chapter 6, page 480). The drawing routine is, of course, placed in Draw( ) and reads as follows:

Listing from "moebius_with_arrows.cpp":

```
P3d P;
V3d x, z;
if ( ShowCaravan )
{
    const Real a = 2;
    int i;
    Real u;
    for ( i = 0, u = U; i < N; i++, u += Delta )
    {
        SetLocalSystem( u, 1, P, x, z, 0, a );
        Arrow.ShadeAtPosition( Yellow, P, x, z );
        SetLocalSystem( u, -1, P, x, z, 0, a );
        Arrow.ShadeAtPosition( Yellow, P, x, z );
    }
}
else
{
    const Real a = 1, b = 2;
    SetLocalSystem( U, State, P, x, z, a, b );
    Arrow.ShadeAtPosition( Yellow, P, x, z );
    SetLocalSystem( U, State, P, x, z, a, -b );
    Arrow.ShadeAtPosition( Red, P, x, z );

    SetLocalSystem( U, State, P, x, z, -a, b );
    Arrow.ShadeAtPosition( Yellow, P, x, z );
    SetLocalSystem( U, State, P, x, z, -a, -b );
    Arrow.ShadeAtPosition( Red, P, x, z );
}
PrintString( Black, -17, 15, "P...pair mode" );
PrintString( Black, -17, 14, "C...caravan mode" );
```

It consists of several calls to SetLocalSystem(...) and the ShadeAtPosition(...) of *Arrow3d*. Depending on the value of the *Boolean* ShowCaravan, the number of calls as well as the input parameters vary. The last two lines of the above listing already indicate that you can switch to pair mode and caravan mode by pressing **P** and **C**, respectively:

Listing from `"moebius_with_arrows.cpp"`:

```
void Scene::Animate( )
{
   int key = TheKeyPressed( );
   if ( key == 'C' )
   {
      ShowCaravan = true;
      Umax = U1 + Delta;
      U = U1;
   }
   if ( key == 'P' )
   {
      ShowCaravan = false;
      Umax = -U1;
   }

   U += 0.05;
   if ( U > Umax )
   {
      U = U1;
      State *= -1;
   }
}
```

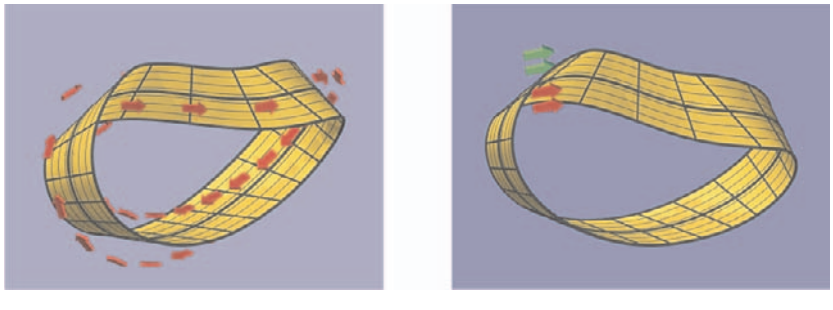

FIGURE 4.27. Two pairs of arrows and a caravan of arrows traveling along a Möbius band.

In Animate( ) we check whether **P** or **C** has been pressed and change DrawCaravan and Umax. Furthermore, we reset U in order to prevent a "traffic jam" of arrows. In the last couple of lines we increase U, reset it to U1 if it is out of range, and, at the same time, switch State. ♢

Example 4.12. Reflecting oloid
We present another example where the remarkable oloid ("wobbler") plays an important part (compare the examples in [14]): An oloid Ω made of a reflecting material and performing the typical "oloid motion" as displayed in `"oloid.cpp"` casts water like caustics on a wall.[16]

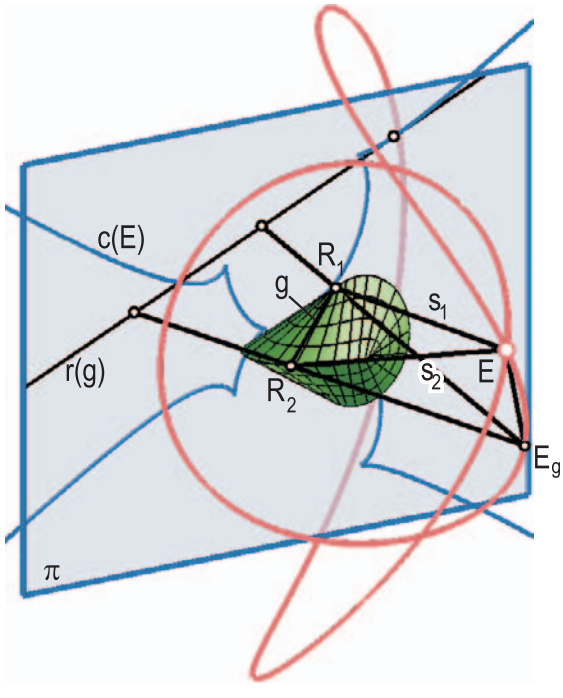

FIGURE 4.28. The caustic curve generated by the reflection on an oloid.

In this example many of Open Geometry's favored tasks are combined: We shade a parameterized surface, perform a complex animation, and calculate an ordinary curve as well as a class curve by direct geometric construction (i.e., without using a parameterized equation).

The oloid can be described as the convex hull of two circles in perpendicular planes such that the center of one circle lies on the circumference of the other.

[16]The idea for this stems from one of our students. He really built a model of the mechanism and presented it at an exhibition.

It is a developable ruled surface (a torse).

This and a few other properties are responsible for the oloid's technical relevance. It is used in mixing and stirring devices. There it performs a special "wobbling" motion that is induced by a relatively simple mechanism. In our example we use this motion to create water caustics on a wall.

First let us have a quick glance at the geometric background: We start with a point light source E and a projection plane π. A ray of light s_1 emitted from E hits the oloid Ω in a point R_1 and is reflected at the tangent plane ε. The reflected ray runs through the reflection E_g of E on ε.

Now we know that Ω is a torse; i.e., it consists of straight lines, and the plane of tangency is constant along each generating line. The reflected rays along a generating line g lie in the plane $\varrho(g) := \mathsf{E}_g$ (E_g is the reflection of E on the tangent plane τ_g along g). Intersecting $\varrho(g)$ with the projection plane π yields a straight line $r(g)$. The one-parameter set $r(g)$ of straight lines envelops a curve $c(\mathsf{E})$ along which the intensity of the reflected light is maximal: a caustic.

Instead of reflecting the rays through E and τ_g we can, of course, project g from E_g. Therefore, the locus $o(\mathsf{E})$ of all points E_g is of interest. It is called the *orthonomic of* Ω *with respect to* E. In our first example (`"reflecting_oloid1.cpp"`) we will construct the orthonomic $o(\mathsf{E})$ and the corresponding caustic $c(\mathsf{E})$ for a given configuration of E, Ω, and π.

We define E and $\pi =: \mathsf{ProjPlane}$ as global constants and take the parameterization of the oloid from `"oloid.cpp"`. There, the oloid is split into four congruent quarters. Consequently, we will define and draw four parts of the orthonomic and four parts of the caustic. We derive four instances $\mathsf{Oloid[i]}$ of the class QuarterOloid and define the orthonomic $o(\mathsf{E})$:

Listing from `"reflecting_oloid1.cpp"`:

```
class MyQuarterOrtho: public ParamCurve3d
{
public:
  P3d CurvePoint( Real u )
  {
    P3d P = Eye;
    P.Reflect( Oloid[idx].TangentPlane( u, 0 ) );
    return P;
  }
  void Def( Color col, int n, Real umin, Real umax, P3d E, int i )
  {
    Eye = E;
    idx = i;
    ParamCurve3d::Def( col, n, umin, umax );
  }
```

```
private:
  P3d Eye;
  int idx;
};
MyQuarterOrtho Ortho[4];
```

The light source E and the index of the oloid part have to be passed as arguments to Def(...). The rest is just the geometric construction of the orthonomic. The caustic's implementation is slightly trickier.

Listing from `"reflecting_oloid1.cpp"`:

```
class MyQuarterCaustic: public ParamCurve3d
{
public:
  P3d CurvePoint( Real u )
  {
    const Real eps = 1e-3;
    Plane p1( Ortho[idx].CurvePoint( u - eps ),
              Oloid[idx].SurfacePoint( u - eps, 0 ),
              Oloid[idx].SurfacePoint( u - eps, 1 ) );
    Plane p2( Ortho[idx].CurvePoint( u + eps ),
              Oloid[idx].SurfacePoint( u + eps, 0 ),
              Oloid[idx].SurfacePoint( u + eps, 1 ) );
    return pi * ( p1 * p2 );
  }
  void Def( Color col, int n, Real umin, Real umax, Plane p, int i )
  {
    pi = p;
    idx = i;
    ParamCurve3d::Def( col, n, umin, umax );
  }
private:
  Plane pi;
  int idx;
};
MyQuarterCaustic Caustic[4];
```

We intersect the projection plane with the planes through a curve point of $o(\mathsf{E})$ and two neighboring generators of the corresponding ruling on Ω. In Init( ) we define the surfaces and curves and we adjust a frame rectangle in the projection plane. In Draw( ) we will shade this rectangle with opacity in order to produce a picture similar to Figure 4.28. In `"reflecting_oloid1.cpp"` we do not, however, draw the straight lines that illustrate the construction of $c(\mathsf{E})$.

Listing from `"reflecting_oloid1.cpp"`:

```
void Scene::Init( )
{
  int n1 = 50, n2 = 5;
  Oloid[0].Def( Green, n1, n2, 1, 1 );
  Oloid[1].Def( Green, n1, n2, -1, 1 );
  Oloid[2].Def( Green, n1, n2, 1, -1 );
  Oloid[3].Def( Green, n1, n2, -1, -1 );
  int i;
  for ( i = 0; i < 4; i++ )
  {
    Ortho[i].Def( Red, 100, 0, 2 * PI / 3 + 1e-4, E, i );
    Caustic[i].Def( Orange, 100, 0, 2 * PI / 3 + 1e-4,
      ProjPlane, i );
    Oloid[i].PrepareContour( );
  }
  const Real x = 20, y = 30;
  Frame.Def( LightBlue, x, y, FILLED );
  Frame.Translate( -0.5 * x, -0.5 * y, 0 );
  V3d n = ProjPlane.n( );
  StrL3d axis( Origin, n ^ Zdir );
  Frame.Rotate( axis, Deg( n.Angle( Zdir ) ) );
  Frame.Translate( ProjPlane.GetPoint( ) );
}
```

Now we want to add the oloid motion to our program. In the sample file `"reflecting_oloid2.cpp"` we copy the class QuarterOloid from `"oloid.cpp"`. In contrast to the previous example it is equipped with a number of methods in order for it to perform the typical "oloid motion." We eliminate a few unnecessary lines from `"oloid.cpp"` and add the orthonomic and caustic curves. We must, however, pay attention to the fact that the position of E and the projection plane π relative to Ω change during the motion. In `"oloid.cpp"` the oloid motion is realized through a point transformation by matrix multiplication. At the beginning of Draw( ) the line MoveAll(1) transforms the oloid to the current position. At the end of Draw( ) the line MoveAll(−1) overwrites all transformed points plus normal vectors with the points and normals of a standard oloid (Proto[0],...,Proto[3]). The corresponding commands for the point and vector transformation read as follows:

Listing from `"reflecting_oloid2.cpp"`:

```
inline void transform( P3d &P )
{
   P3d Q = P − LocalOrigin;
   Q.Rotate( InvR, P );
}
inline void transform( V3d &P )
{
   V3d Q = P;
   Q.Rotate( InvR, P );
}
```

We have to apply the inverse transformation to the light source E and projection plane π, compute orthonomic and caustic, and transfer them again. For this task we can use the global rotation matrix R. We introduce the following inline functions:

Listing from `"reflecting_oloid2.cpp"`:

```
inline void inv_transform( P3d &P )
{
   P.Rotate( R, P );
   P += LocalOrigin;
}
inline void inv_transform( V3d &P )
{
   V3d Q = P;
   Q.Rotate( R, P );
}
```

Now the curve point functions of MyQuarterOrtho and MyQuarterCaustic need some changes:

Listing from "reflecting_oloid2.cpp":

```
P3d CurvePoint( Real u )
{
    P3d P = Eye;
    inv_transform( P );
    P.Reflect( Proto[idx].TangentPlane( u, 0 ) );
    transform( P );
    return P;
}


P3d CurvePoint( Real u )
{
    const Real eps = 1e-3;
    P3d P = ProjPlane.GetPoint( );
    V3d n = ProjPlane.n( );
    P3d Q1 = Ortho[idx].CurvePoint( u - eps );
    P3d Q2 = Ortho[idx].CurvePoint( u + eps );
    inv_transform( P );
    inv_transform( n );
    inv_transform( Q1 );
    inv_transform( Q2 );
    Plane p( P, n );
    Plane p1( Q1, Proto[idx].SurfacePoint( u - eps, 0 ),
              Proto[idx].SurfacePoint( u - eps, 1 ) );
    StrL3d s = p * p1;
    Plane p2( Q2, Proto[idx].SurfacePoint( u + eps, 0 ),
              Proto[idx].SurfacePoint( u + eps, 1 ) );
    P3d S = p * ( p1 * p2 );
    transform( S );
    return S;
}
```

Finally, we move the definition of Ortho[i] and Caustic[i] to the Draw( ) part in order to get the animation. ♢

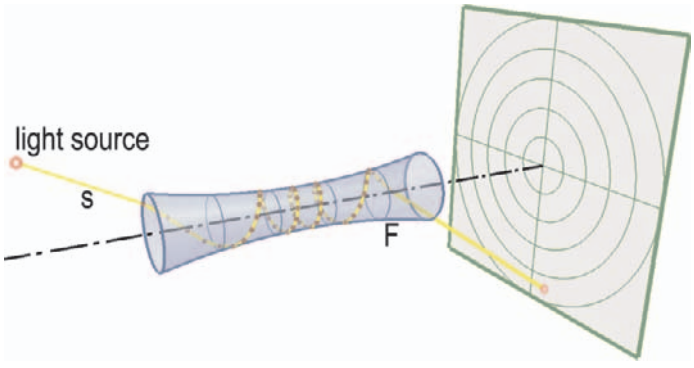

FIGURE 4.29. A ray of light undergoes multiple reflections inside a hyperbolically curved fiber. As the number of reflection approaches ∞, the reflection polygon converges to a geodetic line on the fiber surface.

Example 4.13. Multiple reflection
In technical applications from the field of laser optics and telecommunications it is an important task to send a ray of light r through a small fiber. The fibers are made of glass or some other transparent material of high refractive index. Light traveling inside the fiber center strikes the outside surface at an angle of incidence greater than the critical angle, so that all the light is reflected toward the inside of the fiber without loss. Thus, light can be transmitted over long distances by being reflected inward thousands of times.

Sometimes it is desirable to achieve a high number of inward reflections on a relatively short distance. For that purpose one can use fibers that have the shape of a surface of revolution with only hyperbolic surface points.. However, there is a problem: Certain rays of light will be able to enter the fiber but — after a number of reflections — are sent back in the direction of their origin. It is desirable to know in advance (before producing the special fiber) about the critical directions. Therefore, a computer model is of interest (`"mult_refl.cpp"`). Additionally, we will be able to demonstrate an interesting effect that occurs when the number of reflections approaches ∞.

First we have to design the fiber Φ. It is sensible to model a surface of revolution. Furthermore, we can make good use of an implicit equation $F(x, y, z) = 0$ of Φ.

Our first task will be to intersect a straight line $s \ldots \vec{x}(t) = \vec{p} + t\vec{d}$ with the surface. We can do this by inserting the coordinates of $\vec{x}(t)$ in $F(x, y, z) = 0$ and solving the resulting equation for t. The smallest positive solution will be relevant for our purpose because it belongs to the first intersection point P that is situated on the positive half ray. Having computed $P \ldots \vec{p} = (x_p, y_p, z_p)$ we can at once determine the tangent plane of Φ in P, since its normal vector has direction $\big(F_x, F_y, F_z\big)$.

It will be possible to solve the equation $F\big(\vec{x}(t)\big) = 0$ very rapidly and efficiently if it is algebraic of order four. This is the case if we use a parabola m as meridian curve. If the y-axis of our coordinate system is the axis of Φ, the necessary equations are

$$m \ldots z = Ay^2 + B,\ x = 0 \quad \text{and} \quad \Phi \ldots x^2 + z^2 = (Ay^2 + B)^2.$$

With their help we can implement the fiber surface and a function that returns the tangent plane in a surface point P:

Listing from `"mult_refl.cpp"`:

```
Plane TangentPlane( P3d &P )
{
    const Real ny = -4 * A * A * P.y * P.y * P.y -
                    4 * A * B * P.y;
    return Plane( P, V3d( 2 * P.x, ny, 2 * P.z ) );
}

class MyFiber: public ParamSurface
{
public:
    virtual P3d SurfacePoint( Real y, Real phi )
    {
        P3d P( 0, y, A * y * y + B );
        P.Rotate( Yaxis, phi );
        return P;
    }
};
MyFiber Fiber;
```

Additionally, we need a function to compute the point of reflection P. It needs some computation to find the coefficients of the resulting equation of order four:

Listing from `"mult_refl.cpp"`:

```
Boolean ReflectionPoint( const StrL3d &s, P3d &P )
{
    P3d S = s.GetPoint( );
    V3d dir = s.GetDir( );
    const Real y2 = S.y * S.y;
    const Real d2 = dir.y * dir.y;
    const Real f2 = A * A;
    Real coeff[5];
    coeff[0] = -pow( A * y2 + B, 2 ) + S.x * S.x + S.z * S.z;
    coeff[1] = -4 * f2 * y2 * S.y * dir.y - 4 * A * B * S.y * dir.y +
```

```
            2 * S.x * dir.x + 2 * S.z * dir.z;
    coeff[2] = -6 * f2 * y2 * d2 - 2 * A * B * d2 +
            dir.x * dir.x + dir.z * dir.z;
    coeff[3] = -4 * f2 * S.y * d2 * dir.y;
    coeff[4] = -f2 * d2 * d2;
    Real t[4];
    int n;
    n = ZerosOfPoly4( coeff, t );
    if ( n )
    {
      int i;
      for ( i = 0; i < n; i++ )
      {
        if ( t[i] > 1e-4 )
        {
          P = s.InBetweenPoint( t[i] );
          if ( P.y > Y0 - 0.001 && P.y < Y1 + 0.001 )
            return true;
        }
      }
      return false;
    }
    return false;
  }
```

The input parameters are a straight line s and a point P. If a reflection occurs, the point of reflection will be stored in P and the return value is true. Otherwise, we return false. Note that we assume s to be a model of an oriented light ray; i.e., the light ray starts at the point returned by OPEN GEOMETRY's s.GetPoint( ) and has the direction of s.GetDir( ).

We compute the coefficients of the equation of order four and solve it. The result is stored in the real array t[4]. It is a very convenient property of ZerosOfPoly4(...) that the solutions are stored in *ascending order*. This makes it easy to determine the relevant value t_i. It is the minimal value that is:

- larger than 0 (or a certain small $\varepsilon > 0$) and
- leads to a surface point P with y-coordinate inside a certain interval $[Y_0, Y_1]$ that determines the length of the fiber.

A small but important routine is the following:

Listing from "mult_refl.cpp":

```
Boolean ReflectOnFiber( StrL3d &s, P3d &P )
{
    if ( ReflectionPoint( s, P ) )
    {
        s.Reflect( TangentPlane( P ) );
        s.Def( P, s.GetDir( ) );
        return true;
    }
    else
        return false;
}
```

We compute the reflection point and reflect the straight line s at the tangent plane. Note that it is absolutely necessary to redefine s. If we omit doing this, the next reflection point will be wrong (usually, it will be P again). The final reflection routine traces the path of the incoming ray through the fiber, draws the reflection polygon, and marks the reflection points:

Listing from "mult_refl.cpp":

```
void TraceRay( StrL3d &s )
{
    Real count = 0;
    P3d P, P_old = s.GetPoint( );
    P_old.Mark( Red, 0.2, 0.1 );
    while ( ReflectOnFiber( s, P ) && count < 1000 )
    {
        StraightLine3d( PureYellow, P_old, P, MEDIUM );
        P.Mark( DarkBrown, 0.1 );
        P_old = P;
        count++;
    }
    P_old = ScreenPlane * s;
    Real param = s.GetParameter( P_old );
    if ( param > 0 )
    {
        s.Draw( PureYellow, 0, param, THICK );
        P_old.Mark( Red, 0.2, 0.1 );
    }
        s.Draw( PureRed, 0, 20, MEDIUM );
}
```

The first part is obvious and easy to understand. In the final lines we intersect the outgoing ray with a screen plane that is positioned just after the end of the fiber and mark the intersection point. If the ray is reflected backwards, we do not intersect it with the screen, and we draw it in pure red. The screen is defined in Init( ). It consists of a frame rectangle (Screen) and several circles (Cicrcle[i] that will be drawn on the frame:

Listing from `"mult_refl.cpp"`:

```
const Real l = 16, w = 16;
Screen.Def( LightGray, l, w, FILLED );
Screen.Translate( -0.5 * l, -0.5 * w, 0 );
Screen.Rotate( Xaxis, 90 );
Screen.Translate( 0, Y0 - 8, 0 );

ScreenPlane.Def( Screen[1], Screen[2], Screen[3] );

int i;
Real r, phi = Screen[1].x / CircNum;
for ( i = 0, r = phi; i < CircNum; i++, r += phi )
   Circle[i].Def( DarkGreen, 0.5 * ( Screen[1] + Screen[3] ), Ydir, r, 50 );
```

With the help of the above routines (and a small routine to print certain instructions on the screen) we can write a neat and concise Draw( ) part:

Listing from `"mult_refl.cpp"`:

```
void Scene::Draw( )
{
   Yaxis.LineDotted( Black, -12, 12, 18, MEDIUM );

   Fiber.Contour( DarkBlue, MEDIUM );
   Fiber.VLines( DarkBlue, 6, THIN );
   Fiber.VLines( DarkBlue, 2, MEDIUM );

   StrL3d s;
   s.Def( Source, Fiber.SurfacePoint( Y, Phi ) );
   StraightLine3d( LightYellow, Source, ScreenPlane * s, THIN );
   TraceRay( s );

   Screen.ShadeWithContour( DarkGreen, THICK );
   StraightLine3d( DarkGreen, 0.5 * ( Screen[1] + Screen[2] ),
                   0.5 * ( Screen[3] + Screen[4] ), THIN );
   StraightLine3d( DarkGreen, 0.5 * ( Screen[1] + Screen[4] ),
                   0.5 * ( Screen[2] + Screen[3] ), THIN );
```

```
    Circ3d circle;
    int i, n = 5;
    Real r, phi = Screen[1].x / n;
    for ( i = 0, r = phi; i < n; i++, r += phi )
    {
       circle.Def( DarkGreen, 0.5 * ( Screen[1] + Screen[3] ), Ydir, r, 50 );
       circle.Draw( THIN );
    }

     SetOpacity( 0.5 );
    Fiber.Shade( SMOOTH, REFLECTING );
    SetOpacity( 1 );

     PrintText( -20, 10.8 );
}
```

For the definition of the light ray s we use the global variables

```
P3d Source;
Real Y, Phi;
```

Here Y and Phi are the surface coordinates of the first point of reflection. Source is the point that emits the ray of light. We draw the initial ray until it hits the screen. This will help to get some orientation concerning its direction. Furthermore, we shade the fiber surface with some transparency because the really interesting things will be happening inside.

So far we have done enough to get a nice picture. However, it will be useful to vary Source, Y, and Phi a little. We provide this possibility in Animate( ):

Listing from `"mult_refl.cpp"`:

```
void Scene::Animate( )
{
   int key = TheKeyPressed();
   switch (key )
   {
   case 's':
      Source.z -= 0.05;
      break;
   case 'S':
      Source.z += 0.05;
   break;
   case 'f':
      Y += 0.2;
      break;
```

```
    case 'F':
      Y -= 0.2;
    break;
    case 'j':
      Phi += 1;
      break;
    case 'J':
      Phi -= 1;
    break;
    }
  }
```

Because of the rotational symmetry of the frame it is possible to restrict the light source to $[y, z]$. You can now experiment and vary the incoming ray. It will either pass through the fiber or be cast back. Usually only a few reflections occur. However, if the incoming ray intersects the surface in a very small angle, the number of reflections may rapidly increase (even the maximum number of 1000 reflections as implemented in TraceRay(...) may be exceeded). Perhaps you can reproduce a situation similar to that displayed in Figure 4.29. As the number of reflections increases the reflection polygon converges to a certain curve $c \subset \Phi$.

It is not difficult to see that c is a *geodetic line on* Φ: Two consecutive sides of the reflection polygon are contained in a normal plane ν of Φ. In the passage to the limit, this plane converges to an osculating plane σ of c. Hence the osculating planes of c are normal planes of Φ, which is a characterization of geodetic lines. ♢

5

Open Geometry and Some of Its Target Sciences

With Open Geometry 2.0 it is possible to illustrate concepts from several fields of geometry. In this chapter we present a few examples from

- Descriptive Geometry,
- Projective Geometry,
- Differential Geometry.

Our list of examples is by far not complete, and many more sample files are waiting to be written. We intend only to give you an idea of what is possible with Open Geometry 2.0.

5.1 Open Geometry and Descriptive Geometry

Descriptive geometry is taught in many high school and university courses. Beside teaching basic concepts of construction and drawing, a major aim is the development of spatial thinking. We believe that 3D animations support this to a great extend. Consequently, the development of short movie sequences for the classroom with Open Geometry 2.0 is at the center of our interest in the following. We begin with a very simple example concerning the different views that are used for technical drawing:

Example 5.1. Top, front, and side view
In descriptive geometry it is a common technique to work with top, front, and side views of an object. These views are arranged in a certain position on the drawing plane (a sheet of paper, the computer screen). There, important distances or angles can be measured directly. The first to exploit this configuration of views for constructive purposes in a scientific way was the French mathematician G. MONGE. He explained their genesis in the following way (Figure 5.1):

- Start with three pairwise orthogonal planes π_1, π_2, and π_3 (*image planes*).
- Project the space points orthogonally onto the plane π_i $(i = 1, 2, 3)$.
- Rotate the plane π_j about the axis $\pi_j \cap \pi_2$ through an angle of $90°$ $(j = 1, 3)$.

In the final position the planes π_1 and π_3 coincide with π_2, and we can interpret this plane as the drawing plane.

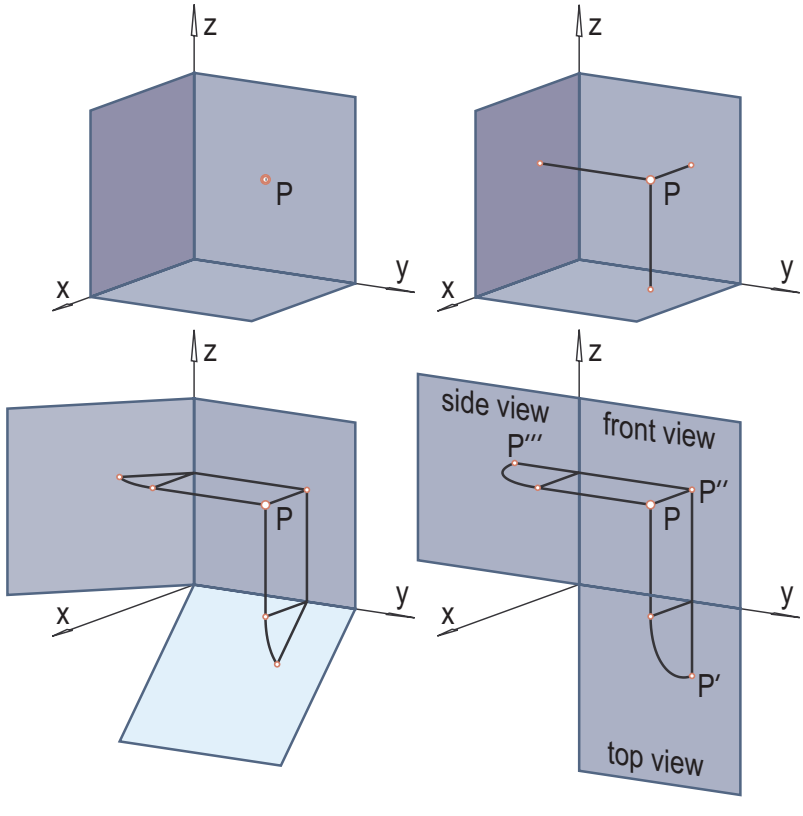

FIGURE 5.1. The genesis of top, front, and side views according to G. MONGE.

The program "views.cpp" yields an animated version of this concept. In the first stage we project a space point P simultaneously onto the three image planes; in the second stage we rotate these planes. The third stage consists of a little pause in order to allow leisurely contemplation of the scene before everything starts again. The following global variables are needed for the program:

Listing from "views.cpp":

```
int Count;
const int N1 = 50;
const int N2 = 100;
const int N3 = 50;

const P3d P( 4, 7, 6 );
P3d Q[3];
P3d R[2];
Sector3d Sector[3];

const Real Side = 10;
Rect3d XYframe, YZframe, ZXframe;
```

The constants N1, N2, and N3 denote the number of frames reserved for each stage. Count takes values between 0 and N1+N2+N3 and gives the current stage of the animation, P is the constant space point; Q[0], Q[1], and Q[2] will wander on the projection rays through P, while R[0] and R[1] are situated on the path circles of the projected points during the rotation. Finally, we will visualize the image planes by three rectangular frames. In Init( ) we bring the points Q[i] and R[i] into their correct position with respect to P:

Listing from "views.cpp":

```
void Scene::Init( )
{
   Count = 0;
   Q[0] = P, Q[1] = P, Q[2] = P;
   R[0].Def( P.x, P.y, 0 );
   R[1].Def( P.x, 0, P.z );

   XYframe.Def( Blue, Side, Side, FILLED );
   YZframe.Def( Blue, Side, Side, FILLED );
   YZframe.Rotate( Yaxis, -90 );
   ZXframe.Def( Blue, Side, Side, FILLED );
   ZXframe.Rotate( Xaxis, 90 );

   AllowRestart( );
}
```

In Draw( ) we draw the line segments PQ[i], the arcs connecting Q[i] with R[i] (during the second and third stages), and additional lines to give some orientation. The frame rectangles are shaded with opacity. The most important part of the program is, however, Animate( ):

Listing from `"views.cpp"`:

```
void Scene::Animate( )
{
   Count++;
   Count = Count % ( N1 + N2 + N3 );
   if ( Count == 0 )
      Restart();
   else if ( Count <= N1 ) // first stage (normal projection)
   {
      Q[0].Translate( 0, 0, -P.z / N1 );
      Q[1].Translate( 0, -P.y / N1, 0 );
      Q[2].Translate( -P.x / N1, 0, 0 );
   }
   else if ( Count <= N1 + N2 ) // second stage (rotation)
   {
      R[0].Rotate( Yaxis, 90.0 / N2 );
      XYframe.Rotate( Yaxis, 90.0 / N2 );
      R[1].Rotate( Zaxis, -90.0 / N2 );
      ZXframe.Rotate( Zaxis, -90.0 / N2 );
      Sector[0].Def( Black, Q[0], Yaxis, R[0], 50 );
      Sector[1].Def( Black, Q[1], Zaxis, R[1], 50 );
   }
}
```

We increase Count and compute its value modulo $N1 + N2 + N3$. If it is zero, we use Open Geometry's restart option to start again. During the first stage we translate Q[i] by a small vector v[i]. The length of v[i] ensures that Q[i] will lie in the plane π_i after exactly N1 steps. During the second stage we treat R[i] and the corresponding frame rectangles in an analogous way and redefine the sectors connecting Q[i] and R[i]. ♢

Our next example deals with shades and shadows, a very beautiful field of descriptive geometry.

Example 5.2. Box shadow
Open Geometry does not support the automatic generation of shades and shadows. However, for certain special cases one can implement them "by hand." In fact, there already exist some simple shadow algorithms in Open Geometry that you might consider useful. An example can be seen in `"box_shadow.cpp"`, where we display the shadow cast by a cube on a horizontal plane.

The shadow algorithms we have mentioned work only for polyhedra. We can compute the shadow of a polyhedron on another polyhedron or on a plane. The plotting of the shadow itself is very simple. We define an instance Cube of the class *Box* and transform it into a polyhedron Polyhedron:

Listing from `"box_shadow.cpp"`:

```
Cube.TransformIntoPolyhedron( );
Polyhedron.Def( *Cube.Phdr );
```

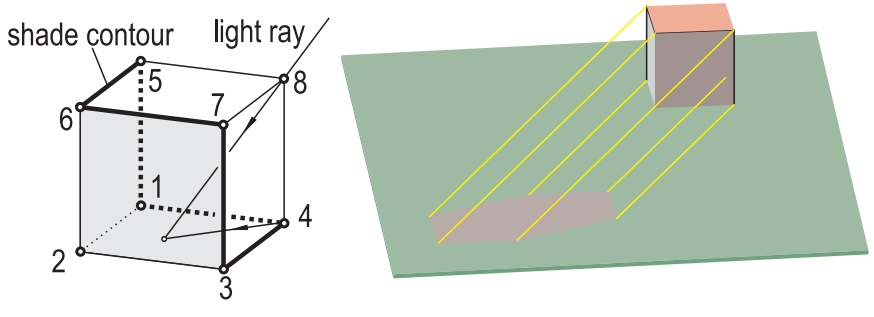

FIGURE 5.2. Shade contour on a cube for parallel light.

In computer graphics, general polyhedra are nasty objects, and their generation is a complex task. Their implementation in Open Geometry is rather tricky too. It makes extensive use of pointers. Whenever possible you should avoid defining a polyhedron directly. Instead, you can rely on predefined Open Geometry methods such as TransformIntoPolyhedron( ). This method exists for the classes *Box*, *RegFrustum*, *FunctionGraph*, and *ParamSurface* (and, consequently, for all derived classes as well).

Now, in order to plot the shadow of the polyhedron in light gray on the horizontal plane $z = 0$, we simply have to write

Polyhedron.PlotShadowOnHorizontalPlane(*LightGray*, 0);

Note that the shadow polygon is not explicitly determined. Internally, OPEN GEOMETRY projects the polyhedron's faces on the plane $z = 0$ and paints them in light gray. This means that we do not have access to the vertices of the shadow polygon or the shade contour on the box. Because this is not too difficult in the case of a cube, we will determine them directly. Without these details the program would lose some of its appeal for, let's say, a demonstration in the classroom.

The global constants and variables we use are the following:

Listing from `"box_shadow.cpp"`:

```
Box Base, Cube;
Polyhedron Polyhedron;

Poly3d CubeFace[5];

const int Index[4][6] = { {2,3,4,8,5,6}, {1,2,3,7,8,5},
                          {1,4,3,7,6,5}, {1,2,6,7,8,4} };
```

We have already mentioned the cube and the polyhedron. Base is the object on which the shadow will be cast, and five faces of the cube will be stored separately as four-sided polygons. We will use them to paint the faces in the shade in a different color. Since in our case the top face will never be in the shade, we need only five faces in the program. The integer array Index[4][6] contains the index sequences for the four different shade contours on the cube that are possible for parallel light from above (compare Figure 5.2). *Box* is a successor of *O3d*, so we can always address Cube's vertices by pPoints[0], pPoints[1],... and need not store or compute the complete shade contour.

Cube and its faces are defined in Init( ):

Listing from `"box_shadow.cpp"`:

```
const Real a = 1.5;
Cube.Def( LightRed, a, a, a );
Cube.Scale( 2 );
Cube.Translate( -a, -a, -a );

int i;
for ( i = 0; i < 5; i++ )
CubeFace[i].Def( LightGray, 4 );

CubeFace[0][1] = Cube.pPoints[1];
CubeFace[0][2] = Cube.pPoints[2];
```

```
CubeFace[0][3] = Cube.pPoints[3];
CubeFace[0][4] = Cube.pPoints[4];
...
CubeFace[4][1] = Cube.pPoints[4];
CubeFace[4][2] = Cube.pPoints[1];
CubeFace[4][3] = Cube.pPoints[5];
CubeFace[4][4] = Cube.pPoints[8];

Cube.Scale( 0.99 );
const Real t = 5;
Cube.Translate( 0, 0, t );
for ( i = 0; i < 5; i++ )
   CubeFace[i].Translate( 0, 0, t );
```

Note the order of the operations: First we center the cube at the origin and use it for the definition of the faces. Then we scale down a little to get the right visibility when shading the cube faces, and *then* we translate everything. If we scaled the cube after translating it, the bottom face would not be visible.

We determine the direction of the incoming light ray in Draw( ). This means that we will be able to change the shadow interactively by manipulating the light source in the Open Geometry window. The four relevant directions can be distinguished according to the sign of the x and y coordinates of the light direction vector. It is globally accessible via LightDirection.L.

Listing from `"box_shadow.cpp"`:

```
int i;
if ( LightDirection.L.x >= 0 )
{
   if( LightDirection.L.y >= 0 )
      i = 0;
   else
      i = 1;
}
else
{
   if( LightDirection.L.y >= 0 )
      i = 2;
   else
      i = 3;
}
```

Now we can draw light rays from the relevant edges of the cube, paint the shade and draw the shade contour. We do this only if the incidence angle of the light rays with respect to the horizontal direction is not too small. The program's output is displayed in Figure 5.2.

Listing from `"box_shadow.cpp"`:

```
if ( LightDirection.L.z > 0.25 )
{
   StrL3d proj_ray;
   P3d P;
   int j;
   for ( j = 0; j < 6; j++ )
   {
      proj_ray.Def( Cube.pPoints[Index[i][j]], LightDirection.L );
      P = XYplane * proj_ray;
      StraightLine3d( PureYellow, Cube.pPoints[Index[i][j]],
                      P, MEDIUM );
   }
   Polyhedron.PlotShadowOnHorizontalPlane( LightGray, 0 );

   CubeFace[0].Shade( );
   if ( i == 0 )
   {
      CubeFace[1].Shade( );
      CubeFace[4].Shade( );
   }
   if ( i == 1 )
   {
      CubeFace[3].Shade( );
      CubeFace[4].Shade( );
   }
   if ( i == 2 )
   {
      CubeFace[1].Shade( );
      CubeFace[2].Shade( );
   }
   if ( i == 3 )
   {
      CubeFace[2].Shade( );
      CubeFace[3].Shade( );
   }
   for ( j = 0; j < 5; j++ )
      StraightLine3d( DarkGray, Cube.pPoints[Index[i][j]],
                      Cube.pPoints[Index[i][j+1]], THIN );
   StraightLine3d( DarkGray, Cube.pPoints[Index[i][5]],
                   Cube.pPoints[Index[i][0]], THIN );
}
```

♢

The following example deals with a more advanced topic of descriptive geometry. a *nonlinear projection.*

Example 5.3. Net projection
In descriptive geometry there exist several types of projections. The most common are central projection, parallel projection, and normal projection. From the point of view of projective geometry, these three types are equivalent: Given an eye point E (either a finite point or an ideal point) and an image plane π with $E \notin \pi$, the projection p is defined as

$$p\colon \mathbb{P}_3 \setminus \{E\} \to \pi, \qquad P \mapsto [E, P] \cap \pi.$$

In this formula $\mathbb{P}_3$ denotes real projective 3-space. However, there exist other types of projection: For example, let $\mathcal{L}$ be a line congruence (a two-parameter set of straight lines) such that a generic space point P is incident with only one line $l_P \in \mathcal{L}$. We can use the lines of $\mathcal{L}$ as projection rays and obtain the map

$$q\colon \mathbb{P}_3 \setminus D \to \pi, \qquad P \mapsto l_P \cap \pi.$$

The set of exceptional points D is of dimension two or even lower. Of course, the properties of these projections may differ considerably from the ordinary *linear* projections. For example, the image of a straight line is usually curved.

We present an example of a projection of that kind: the *net projection* n. The set $\mathcal{L}$ of projection rays consists of all straight lines that intersect two skew straight lines f_1, f_2: the *focal lines* (Figure 5.3).

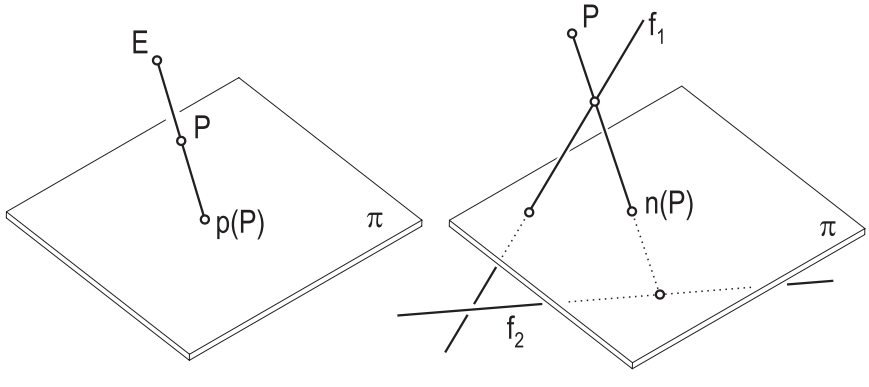

FIGURE 5.3. Central projection and net projection.

The projection n is defined for all points $P \in \mathbb{P}_3 \setminus \{f_1, f_2\}$. The unique projection ray through P can be determined by intersecting the planes $[P, f_1]$ and $[P, f_2]$. The implementation of the net projection in Open Geometry is quite obvious (`"net_projection1.cpp"`). We write a function NetProject(...) that projects a space point onto $[x, y]$:

Listing from "net_projection1.cpp":

```
P3d Project( const P3d &P, const StrL3d &f1,
   const StrL3d &f2 )
{
   Plane p1( P, f1 );
   Plane p2( P, f2 );
   StrL3d s = p1 * p2;
   return XYplane * s;
}
```

Now we want to display the net projection $n(c)$ of some space curve c. We decide on a sine curve drawn on a cylinder of revolution. The curve c has to be implemented as a parameterized curve.

Listing from "net_projection1.cpp":

```
class MyCurve: public ParamCurve3d
{
public:
   P3d CurvePoint( Real u )
   {
      const Real r = 3;
      return P3d( r * cos( u ), r * sin( u ), r * sin( 2 * u ) );
   }
};
MyCurve Curve;
```

The image curve Image $= n(c)$ could be implemented as parameterized curve as well. But in order to show its generation, we will draw it as *PathCurve3d*. Of course, before doing this we must initialize both curves c and $n(c)$ in Init( ). Additionally, we adjust a frame rectangle in $[x, y]$. The global real variable U will serve as parameter for the animation.

Listing from "net_projection1.cpp":

```
void Scene::Init( )
{
  Curve.Def( Red, 150, −PI, PI );
  Image.Def( Green, 1000, 0.1 );

  U = 1.2;

  const Real x = 20, y = 20;
  Frame.Def( Blue, x, y, FILLED );
  Frame.Translate( −0.5 ∗ x, −0.5 ∗ y, 0 );
}
```

For the net projection we use two globally constant straight lines with the z-axis as common normal.

Listing from "net_projection1.cpp":

```
P3d NetProject( const P3d &P, const StrL3d &f1,
  const StrL3d &f2 )
{
  Plane p1( P, f1 );
  Plane p2( P, f2 );
  StrL3d s = p1 ∗ p2;
  return XYplane ∗ s;
}
```

In Draw( ) we compute the new curve point of $c(n)$ and draw the part within the frame rectangle.

Listing from "net_projection1.cpp":

```
P3d C = Curve.CurvePoint( U );
P3d I = NetProject( C, Line1, Line2 );
if ( fabs( I.x ) < Frame[3].x && fabs( I.y ) < Frame[3].y )
  Image.AddPoint( I );
Image.Draw( MEDIUM, STD_OFFSET, 2 );
```

There are four points of interest on the projection ray through a curve point $C \in c$: the curve point itself, its image point I, and the intersection points L_1 and L_2 with the focal lines f_1 and f_2, respectively. We ensure that the whole line segment between them is displayed in a medium line style by drawing the three segments through C. The result can be seen in Figure 5.4. The projection $n(c)$ of c is an attractive curve with four axes of symmetry.

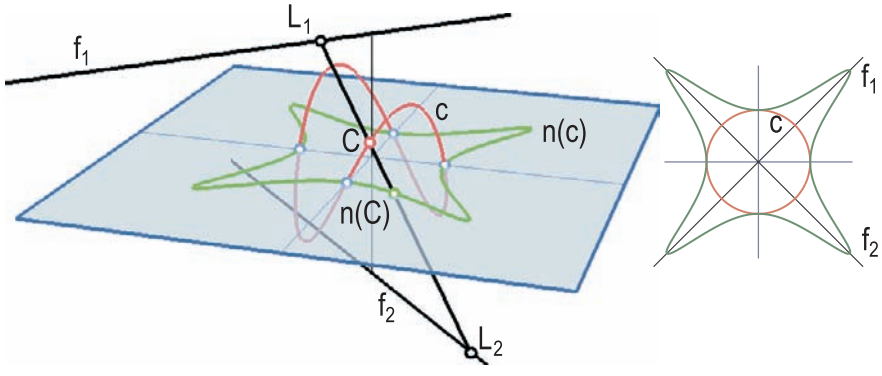

FIGURE 5.4. The net projection $n(c)$ of a space curve c. The image on the right-hand side displays a top view.

A net projection does not necessarily need two real focal lines. It is possible for f_1 and f_2 to be *coincident* or *conjugate complex*. The first case leads to a *parabolic net projection*, the second to an *elliptic net projection*. The projection used in `"net_projection1.cpp"` is called *hyperbolic*.

In `"net_projection2.cpp"` we use a net of lines that is invariant under the group of rotations about a fixed axis (*nets of rotation*). The focal lines of these nets are necessarily conjugate complex and contain the circular points at infinity I, and $\bar{I}$, of the planes perpendicular to the axis. This implies, of course, that we cannot use the constructive approach to the NetProject(...) function of `"net_projection1.cpp"`.

The equations of these special net projections are needed. We describe them in a Euclidean coordinate system $\{O; x, y, z\}$, where $[x, y]$ is the projection plane and z is the net's axis of rotation. Without loss of generality, we can assume that

$$x + iy = 0, \qquad z = \pm D$$

are the equations of the conjugate complex focal lines. This gives us some idea of the two-parameter manifold of net rays: Consider the helical motion with pitch D and axis z. Then the net rays are the *tangents of the path curves of the points in* $[x, y]$ (Figure 5.5).

There exists exactly one (real) projection ray through a real space point $P \in \mathbb{P}_3 \setminus \{f_1, f_2\}$ with coordinate vector $(p_x, p_y, p_z)^t$. It intersects $[x, y]$ in a point

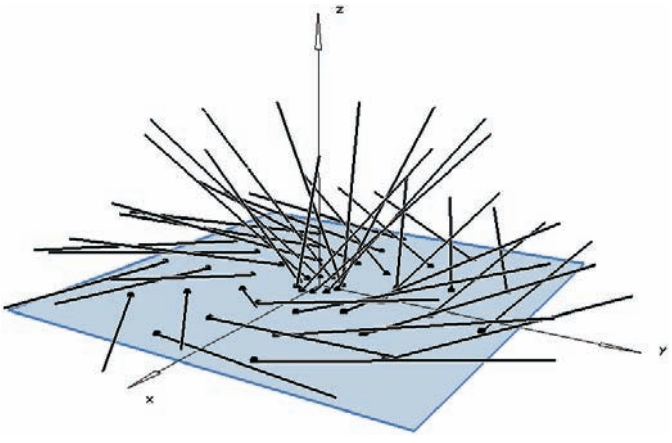

FIGURE 5.5. The lines of an elliptic congruence with rotational symmetry are tangents of the trajectories of a screw motion.

$n(P)$ with coordinates

$$x = \frac{D(Dx + yz)}{D^2 + z^2}, \qquad y = \frac{D(Dy - xz)}{D^2 + z^2}.$$

With these equations we can rewrite the NetProject(...) function:

Listing from `"net_projection2.cpp"`:

```
P3d NetProject( const P3d &P, const Real d )
{
   Real denom = d * d  + P.z * P.z;
   return P3d( d * ( d * P.x + P.y * P.z ) / denom ,
               d * ( d * P.y - P.x * P.z ) / denom, 0 );
}
```

We can then apply it to an arbitrary space curve c. For Figure 5.6 we chose a circle with axis x in $[\,y, z\,]$. Its image b is a *lemniscate of* BERNOULLI.

The surface Π of projection rays through the circle c is quite attractive. It is a ruled surface of order four, since it consists of all straight lines that intersect the two focal lines and a circle (you have to sum up the orders of these curves to get the order of Π: $1 + 1 + 2 = 4$). We visualize it in `"net_projection3.cpp"`. Its implementation simply reads as follows:

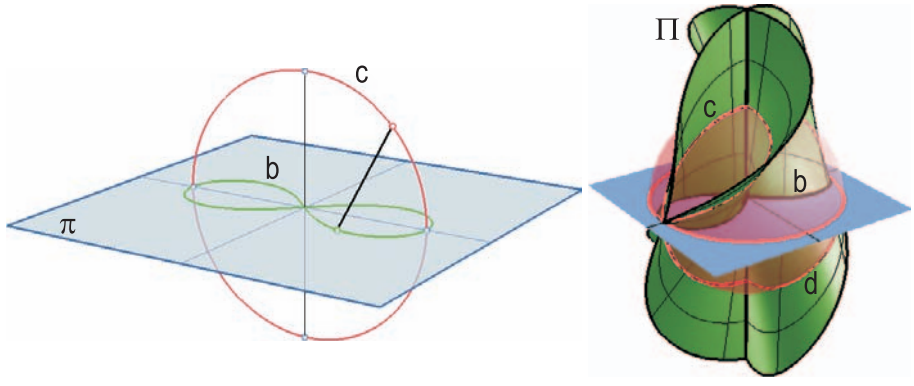

FIGURE 5.6. Projecting a circle c by the rays of a net of rotation.

Listing from `"net_projection3.cpp"`:

```
class MyRuledSurface: public RuledSurface
{
   virtual P3d DirectrixPoint( Real u )
   {
      return Curve.CurvePoint( u );
   }
   virtual V3d DirectionVector( Real u )
   {
      P3d P = Curve.CurvePoint( u );
      return V3d( P, NetProject( P, D ) );
   }
};
MyRuledSurface Surface;
```

Rotating c about z yields a sphere S. If the radius of c is equal to the pitch D of the helical motion that defines the net rays, the intersection curve of Π and S splits up into the circle c, the two focal lines f_1 and f_2, and a space curve d of order four. We display these curves together with S. The curve d can be obtained by reflecting the points of c on their respective projection images. Hence, we can implement it in the following way:

Listing from "net_projection3.cpp":

```
class MyCurveOfOrder4: public ParamCurve3d
{
public:
   P3d CurvePoint( Real u )
   {
      P3d I = Image.CurvePoint( u );
      return I + V3d( Curve.CurvePoint( u ), I );
   }
};
MyCurveOfOrder4 CurveOfOrder4;
```

In order to see what is happening inside, we draw the sphere S with opacity. The output of the program is displayed in Figure 5.6. ♢

5.2 Open Geometry and Projective Geometry

Projective geometry may be called the *mother of geometry* (although there are a few geometries that are not her offspring). It deals with points, straight lines, planes, and their mutual incidence. Concepts like "distance," "angle," "midpoint," and "parallel" cannot be defined within pure projective geometry. They belong to Euclidean geometry or her elder sister, affine geometry.

The foundations of projective geometry date back very far. As usual, already the ancient Greeks possessed a certain knowledge about *points at infinity* and proved theorems of a projective nature. In the Middle Ages it was mainly the art of painting and architecture that promoted projective geometry. It is very useful for the drawing of perspective images in descriptive geometry.

Later, when the theory had developed further, mathematicians and geometers found that projective geometry permitted very deep insight into certain problems. Many theorems could be stated in a general and elegant form. Compare, for example, a very simple theorem of plane affine geometry and the corresponding projective theorem:

- *Two distinct straight lines a and b have exactly one intersection point or are parallel.*
- *Two distinct straight lines a and b have exactly one intersection point.*

The difference is even more visible in the generalization of the above theorem (Bezout's theorem). Its projective form reads; *Any two algebraic curves of order M and N have either infinitely many or exactly $M \cdot N$ common points.*

We will give a short algebraic introduction to the basic concepts of projective geometry. Afterwards, we will present examples of how Open Geometry can support these concepts. The difference between two- and three-dimensional projective geometry is rather small. So we will immediately start with the spatial case. From time to time we will consider examples from planar geometry as well.

Let $\mathbb{E}_3$ be real Euclidean space of dimension three. To us, a point of $\mathbb{E}_3$ is a triple $(x, y, z)^t$ of real numbers. We define a map from $\mathbb{E}_3$ into the set $\mathbb{P}_3$ of *one-dimensional subspaces* of $\mathbb{R}^4$:

$$(x, y, z)^t \mapsto [(1, x, y, z)^t].$$

$\mathbb{P}_3$ is already the projective space of dimension three. Its "points" are the one-dimensional subspaces of $\mathbb{R}^4$. Any point $X \in \mathbb{P}_3$ is determined by a single, nonzero vector $(x_0, x_1, x_2, x_3)^t \in \mathbb{R}^4$. Proportional vectors determine the same point, i.e., only the *ratio of the coordinates* is of importance. The vector $(x_0 : x_1 : x_2 : x_3)^t$ is called the *homogeneous coordinate vector* of X. The relation between Euclidean and homogeneous coordinates is expressed in the equations

$$x = \frac{x_1}{x_0}, \quad y = \frac{x_2}{x_0}, \quad z = \frac{x_3}{x_0}. \tag{1}$$

The transition from homogeneous coordinates to Euclidean coordinates is, however, possible only if $x_0 \neq 0$. In other words, the projective points described by $x_0 = 0$ do not correspond to Euclidean points. They are *points at infinity* or *ideal points*. All points at infinity lie in a plane ω, the *plane at infinity* described by the equation $x_0 = 0$.

A very lucid geometric interpretation is possible in the planar case. We embed the Euclidean plane $\mathbb{E}_2$ in Euclidean 3-space $\mathbb{E}_3$ by mapping $(x, y)^t$ to $(1, x, y)^t$ (Figure 5.7). Now we consider a point $A(1, a_x, a_y)^t$ of $\mathbb{E}_2$ and draw its connection line a with $O(0, 0, 0)$. Any direction vector of a gives just the homogeneous coordinates $(1 : a_x : a_y)^t$ of A.

However, a straight line b through O with direction vector $(0, b_x, b_y)^t$ does not intersect $\mathbb{E}_2$ in a finite point. According to our considerations, it belongs to a point at infinity B_u. In Figure 5.7 we indicate B_u by a straight line and a little white arrow pointing in its direction.[1]

How can we perform computations in homogeneous coordinates? It is not difficult, but there are two important things to remember:

[1] This geometric interpretation of homogeneous coordinates is closely related to rational Bézier curves and rational B-spline curves (compare page 130). In fact, rational Bézier curves possess *integral* parametric representations in *homogeneous coordinates*. Comparing Figure 2.52 in Chapter 2 and Figure 5.7 makes this evident.

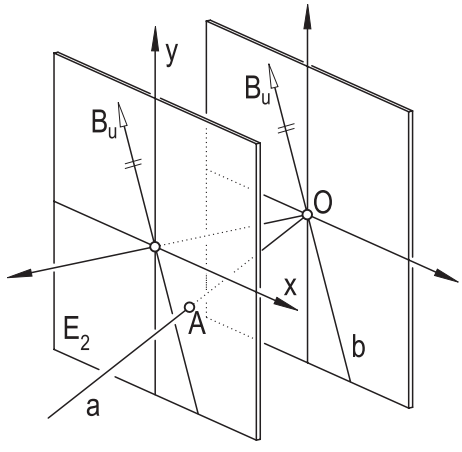

FIGURE 5.7. Extending $\mathbb{E}_2$ to the projective space $\mathbb{P}_2$.

1. The homogeneous coordinates are determined only up to a constant factor; i.e., $(x_0 : x_1 : x_2 : x_3)^t$ and $(\lambda x_0 : \lambda x_1 : \lambda x_2 : \lambda x_3)^t$ describe the same point $X \in \mathbb{P}_3$.

2. At least one coordinate must always be different from zero. No point of $\mathbb{P}_3$ corresponds to $(0:0:0:0)^t$.

We give you a few simple examples in two dimensions. A Euclidean straight line s is described by an equation of the shape $u_0 + u_1 x + u_2 y = 0$. From the 2D version of equation (1) we find its projective form

$$u_0 x_0 + u_1 x_1 + u_2 x_2 = 0. \tag{2}$$

The line at infinity u of $\mathbb{P}_2$ is described by $x_0 = 0$. We can easily compute the intersection point of s with the line at infinity u by inserting $x_0 = 0$ in (2). The intersection point is $S_u(0 : u_2 : -u_1)^t$ – the point at infinity of s. Note that $(u_2, -u_1)^t$ is the direction vector of s.

For a second example we consider the hyperbola described by

$$a^2 b^2 x_0^2 - b^2 x_1^2 + a^2 x_2^2 = 0.$$

It intersects u in two points $(0 : a : \pm b)^t$, the points at infinity of its asymptotes. Things get very strange if we intersect an arbitrary circle

$$(x - m)^2 + (y - n)^2 = r^2$$

with the line at infinity. The conjugate complex intersection points are given by $I, \bar{I}(0:1:\pm i)^t$. They depend neither on the circle's midpoint nor on its radius, i.e.,

all circles intersect in two fixed points at infinity. Conversely, any conic section passing through I and $\bar{I}$ is a circle. For this reason I and $\bar{I}$ are called *circular points at infinity*.

The last example shows that projective geometry can be really enlightening to certain geometric phenomena. Especially the projective theory of conic sections is of outstanding elegance. We used some aspects of it for the implementation of the class *Conic* in Open Geometry. We will not pursue it at this place and turn to other central concepts of projective geometry. The interested reader is referred to the section on conics (Section 2.5 on page 110).

The transformations of Euclidean space are characterized by the invariance of distance. The invariant property of projective transformation is the *collinearity of points*. That is, if κ is a projective transformation and X, Y, and Z are collinear points, then $\kappa(X)$, $\kappa(Y)$ and $\kappa(Z)$ are collinear as well.[2] The group of projective transformations is very large and general. It contains all Euclidean transformations and those transformations that belong to the geometries that are called *non-Euclidean* nowadays. In homogeneous coordinates a projective transformation reads

$$\begin{aligned} x'_0 &= a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + a_{03}x_3, \\ x'_1 &= a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + a_{13}x_3, \\ x'_2 &= a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + a_{23}x_3, \\ x'_3 &= a_{30}x_0 + a_{31}x_1 + a_{32}x_2 + a_{33}x_3. \end{aligned}$$

It is bijective (regular) iff the coefficient matrix $\mathbf{A} = (a_{ij})$ is regular. With the help of this matrix and the abbreviations $\mathbf{x} = (x_0 : x_1 : x_2 : x_3)^t$, $\mathbf{x}' = (x'_0 : x'_1 : x'_2 : x'_3)^t$ we can rewrite the above transformation as

$$\mathbf{x}' = \mathbf{A}\mathbf{x}.$$

This simple and general form makes projective transformation useful for computer graphics, and in fact, it arises very often in OpenGL code.

Invariants of projective transformations are of special importance. We already mentioned the collinearity of points. Another important invariant is the *cross ratio*. We consider four values $s_0, s_1, s_2, s_3 \in \overline{\mathbb{R}} := \mathbb{R} \cup \infty$ and assign a cross ratio

$$\mathrm{CR}(s_0, s_1, s_2, s_3) := \frac{s_0 - s_2}{s_0 - s_3} : \frac{s_1 - s_2}{s_1 - s_3}$$

to them. The cross ratio takes values in $\overline{\mathbb{R}}$ and is well defined if no three of the values s_i are identical. A good and stable implementation of the cross ratio in a computer program has to take care of some special cases. Because of the fundamental importance of the cross ratio in projective geometry, it is really worth the trouble. The following lines stem from Open Geometry's `"proj_geom.h"` file:

[2] For this reason, projective transformations are called *collinear maps* as well.

Listing from "H/proj_geom.h":

```
Boolean SetCrossRatio( Real &ratio, const Real q[4],
                       Real eps = 1e-6, Real infinity = 1e6 )
{
   Real r[4];
   int i;
   for ( i = 0; i < 4; i++ )
   {
      r[i] = q[i];
      if ( fabs( r[i] ) > infinity )
         r[i] = infinity * infinity;
   }

   int j;
   for ( i = 0; i < 4; i++ )
   {
      for ( j = i + 1; j < 4; j++ )
         if ( fabs( r[i] - r[j] ) < eps )
         {
            int k;
            for ( k = j + 1; k < 4; k++ )
            {
               if ( fabs( r[j] - r[k] ) < eps )
                  SafeExit( "identical...!" );
            }
         }
   }

   if ( fabs( r[0] - r[3] ) < eps || fabs( r[1] - r[2] ) < eps )
   {
      ratio = infinity;
      return false;
   }
   else
   {
      ratio = ( r[0] - r[2] ) * ( r[1] - r[3] )
            / ( r[0] - r[3] ) / ( r[1] - r[2] );
      if ( fabs( ratio ) > infinity )
      {
         ratio = infinity;
         return false;
      }
      else
         return true;
   }
}
```

SetCrossRatio(...) returns a value of type *Boolean*: true if the cross ratio is finite and false otherwise. The *Real* variable ratio is set to the actual value of the cross ratio determined by the input values q[0],...,q[3]. In addition, we perform the necessary checks for (almost) infinite and (almost) identical elements.

The cross ratio can be defined only only for real values but also for four points of a straight line or conic. We are not even restricted to points: The lines or planes of a pencil, tangents of a conic, and tangent planes of a cone of second order are other targets for this universal tool.

We start by defining the cross ratio for points of the fixed straight line s defined by $x_2 = x_3 = 0$. Identifying s with the real projective line $\mathbb{P}_1 = \{(x_0 : x_1)^t\}$, we define the *cross ratio* of four points $A, B, C, D \in \mathbb{P}_1$ with homogeneous coordinates $(a_0 : a_1)^t, \ldots, (d_0 : d_1)^t$ as

$$\mathrm{CR}(A, B, C, D) := \frac{\begin{vmatrix} a_0 & c_0 \\ a_1 & c_1 \end{vmatrix}}{\begin{vmatrix} a_0 & d_0 \\ a_1 & d_1 \end{vmatrix}} : \frac{\begin{vmatrix} b_0 & c_0 \\ b_1 & c_1 \end{vmatrix}}{\begin{vmatrix} b_0 & d_0 \\ b_1 & d_1 \end{vmatrix}}.$$

Since we have $\mathrm{CR}(A, B, C, D) = \mathrm{CR}(a_1/a_0, \ldots, d_1/d_0)$, this definition does not depend on the special choice of the homogeneous coordinate vector. We can easily compute the cross ratio of the Euclidean coordinates of the points A, B, C, D. If one of them is the point at infinity, we have to insert the value ∞.

The *cross ratio* of four arbitrary points A', B', C', D' on a straight line s' can be computed in two steps (Figure 5.8):

1. Transform s' to s by a projective transformation κ.
2. Compute the cross ratio of the transformed points $\kappa(A'), \ldots, \kappa(D')$.

It is not difficult to prove that the value of the cross ratio does not depend on the special choice of κ. The invariance of the cross ratio under projective transformations is obvious by definition.

Now to the cross ratio for the elements of a pencil of lines $V_\varepsilon(v)$; V is the vertex, ε the supporting plane of the pencil. We define the cross ratio of four straight lines $a, b, c, d \in V_\varepsilon$ with the help of an arbitrary straight line $s' \subset \varepsilon$ that does not contain V:

$$\mathrm{CR}(a, b, c, d) := \mathrm{CR}(A', B', C', D'),$$

where $A' = a \cap s', \ldots, D' = d \cap s'$ (Figure 5.8). The cross ratio of four planes $\alpha, \beta, \gamma, \delta$ of a pencil of planes with axis v is defined as the cross ratio of the four intersection lines with an arbitrary plane ε that does not contain v. In both cases, the choice of s' or ε has no influence on the value of the cross ratio.

The implementation of the cross ratio of four collinear points is not difficult. Consider the following example:

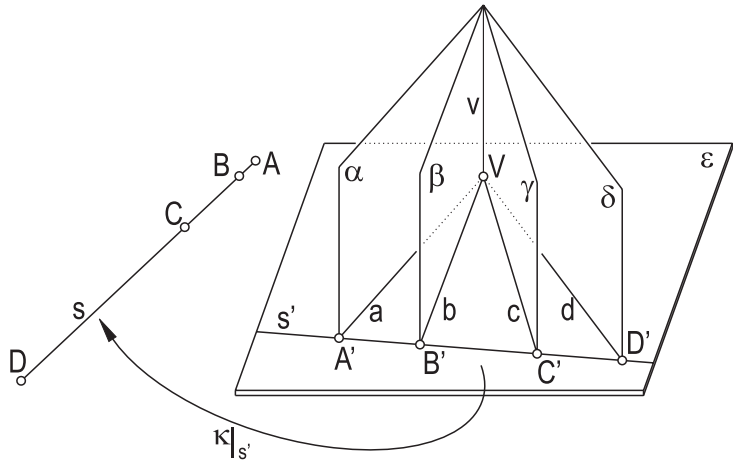

FIGURE 5.8. The cross ratio of four collinear points, four straight lines of a pencil, and four planes with common axis.

Listing from `"H/proj_geom.h"`:

```
Boolean SetCrossRatio( Real &ratio, const P3d P[4],
                       Real eps = 1e-6, Real infinity = 1e6 )
{
   StrL3d s;
   if ( P[0].Distance( P[1] ) < eps )
      s.Def( P[0], P[2] );
   else
      s.Def( P[0], P[1] );
   Real r[4];
   int i;
   for ( i = 0; i < 4; i++ )
      r[i] = ( s.GetParameter( P[i] ) );
   return SetCrossRatio( ratio, r, eps, infinity );
}
```

The implementation of the cross ratio of four straight lines or planes has to be either a little tricky or a little "dirty." We decide on the latter:

Listing from `"H/proj_geom.h"`:

```
Boolean SetCrossRatio( Real &ratio, const Plane pi[4],
                       Real eps = 1e-6, Real infinity = 1e6 )
{
   StrL3d s( P3d( -5.917409748, 2.900873921137, 7.91747412981 ),
             V3d( 0.9747938741, -0.77937911138, 0.19387499483 ) );
   P3d P[4];
   int i;
   for ( i = 0; i < 4; i++ )
      P[i] = pi[i] * s;
   return SetCrossRatio( ratio, P, eps, infinity );
}
```

The cross ratio of four planes is computed as the cross ratio of the intersection points with a "weird" straight line w. If w intersected the axis of the pencil of planes, it would not be valid. But this is extremely unlikely. The remaining cross ratios are implemented in an analogous fashion.

So far, we have not talked about the cross ratio of four points on a conic or four tangent planes of a cone of second order. The cross ratio of four conic points A, B, C, D is defined via the cross ratio of four straight lines a, b, c, d of a pencil $V(v)$ with vertex V on the conic. We simply connect $A, \ldots, D$ with V. If V happens to be one of the four points in question, it corresponds to the conic tangent in V (Figure 5.9).

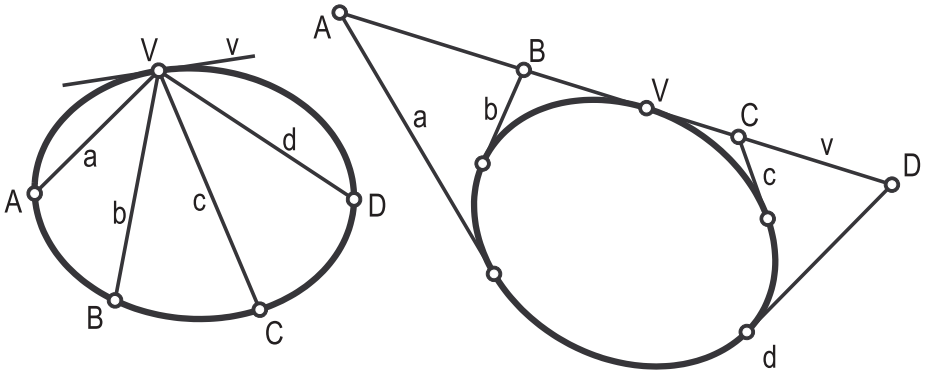

FIGURE 5.9. The cross ratio of four points or tangents of a conic.

The cross ratio of four tangents a, b, c, d of a conic is defined as the cross ratio of the four intersection points A, B, C, D with a fixed tangent v of the conic. If v is one of the tangents a, b, c, d, the intersection point has to be replaced by the point of tangency V of v.

Finally, in order to compute the cross ratio of four tangent planes of a cone of second order, we intersect the tangent planes and the cone itself with an arbitrary plane ε that does not contain the cone's vertex. The four intersection lines and the intersection conic define the cross ratio.

Hence, we have defined a cross ratio for:

- four points of a straight line or conic,
- four straight lines lying in a pencil or tangent to a conic,
- four planes lying in a pencil or tangent to a cone of second order.

A cross ratio may be defined for further geometric objects as well (pencil of conics, points on a rational curve,...), but for the time being we shall leave it at that. If you want to extend the notion of the cross ratio to other objects, you can always rely on the cross ratio of four numbers in $\overline{\mathbb{R}}$.

With the help of the cross ratio we can introduce projective coordinates in an arbitrary set $\mathcal{S}$ of geometric objects where a cross ratio is defined for any four elements of $\mathcal{S}$. For example, $\mathcal{S}$ may be the set of tangent planes of a cone of second order or the set of points on a conic.

Let U, O, and E be three fixed elements of $\mathcal{S}$. We call them *element at infinity*, *origin*, and *unit element*. You may think of the point at infinity, origin, and unit point on the real line. Any element $X \in \mathcal{S}$ defines a cross ratio $\mathrm{CR}(U, O, E, X)$ that, reversed, can be used to identify X unambiguously. In other words, the triple (U, O, E) is a projective coordinate system or *projective scale* on $\mathcal{S}$.

Consider the example of the real line. There, the point X at oriented distance x from the origin O corresponds to the cross ratio $\mathrm{CR}(U, O, E, X) = \mathrm{CR}(\infty, 0, 1, x) = x$. Hence, projective coordinates are a generalization of Euclidean coordinates. They make it possible, however, to assign a coordinate to the point at infinity as well: U belongs (of course) to ∞.

In Open Geometry we implemented the classes *ProjScale* and *ProjScale2d*. The latter applies only to sets of 2D elements with cross ratio (i.e., pencils of points and lines or points and tangents of conics), while the first can be used for 3D elements as well. We will describe the implementation of *ProjScale2d* because it has fewer cases and is more lucid. The class header is listed below.

Listing from `"H/proj_geom.h"`:

```
class ProjScale2d
{
public:
   ProjScale2d( );
   ProjScale2d & operator = ( ProjScale2d &other );
```

```
    void Def( const P2d &V, StrL2d t[3] );
    void Def( const StrL2d &l, P2d Q[3] );
    void Def( Conic &c, P2d Q[3] );
    void Def( Conic &c, StrL2d t[3] );
    void Draw( Color col, Real u0, Real u1, ThinOrThick thick );
    void Draw( ThinOrThick thick );
    void Mark( Color col, Real rad1, Real rad2 = 0 );
    void MarkElement( Color col, Real rad1, Real rad2, int i );
    void MarkElements( Color col, Real rad1, Real rad2 );
    void DrawElement( Color col, Real u0, Real u1,
      ThinOrThick thick, int i );
    void DrawElements( Color col, Real u0, Real u1,
      ThinOrThick thick );
    ProjType2d GetType( ) { return Type; }
    void SetElement( const StrL2d &t, int i )
      { s[i] = t; ComputeReals( ); }
    void SetElement( const P2d &Q, int i )
      { P[i] = Q; ComputeReals( ); }
    P2d GetPoint( int i ) { return P[i]; }
    StrL2d GetLine( int i ) { return s[i]; }
    P2d PointOfCrossRatio( const Real ratio );
    StrL2d LineOfCrossRatio( const Real ratio );
    Real CrossRatio( const P2d &A, const Real eps = 1e-6,
                     const Real infinity = 1e6 );
    Real CrossRatio( const StrL2d &a, const Real eps = 1e-6,
                     const Real infinity = 1e6 );
  private:
    void ComputeReals( );
    StrL2d Line;
    P2d Point;
    Conic conic;
    P2d P[4];
    StrL2d s[4];
    Real u[4];
    ProjType2d Type;
  };
```

To begin with, the private members of the class deserve most interest. We want to implement a very general and easy to use concept of projective scales. Therefore, any instance of the class *ProjScale2d* has the possibility of being either a pencil of points, a pencil of lines, the set of points on a conic, or the set of tangents of a conic. The type is stored in Type, a variable of type *ProjType2d* that is defined in `"proj_geom.h"` as well:

Listing from "H/proj_geom.h":

```
enum ProjType2d { PencilOfLines2d, PencilOfPoints2d,
                  PointsOnConic2d, TangentsOfConic2d };
```

In any case, *ProjScale2d* has four member variables of types *P2d* and *StrL2d*. For necessary computations we will need four variables of type *Real* as well. Each of the four types corresponds to one of the four defining methods. All private member variables are set to certain sensible values, regardless of how the instance of *ProjScale2d* is defined.

The projective scale is defined by P[0], P[1], and P[2] (or s[0], s[1], and s[2]), while P[3] (or s[3]) will be convenient for the CrossRatio(...) method that will be described later on. The three real variables u[0], u[1], and u[2] are set to values that transform the projective scale immediately to the parameter distribution on Line. That is, we always have an original projective scale *and* a projective scale on a straight line. The transformation uses the geometric ideas that we have explained above. You will immediately recognize this if you have a look at the next listing:

Listing from "H/proj_geom.h":

```
void ProjScale2d::ComputeReals( )
{
   if ( Type == PencilOfPoints2d )
   {
      u[0] = Line.GetParameter( P[0] );
      u[1] = Line.GetParameter( P[1] );
      u[2] = Line.GetParameter( P[2] );
   }
   else if ( Type == PencilOfLines2d )
   {
      Line.Def( P2d( -2.1374109387438,  5.008010107416 ),
                V2d(  0.9187413872982, -0.791873444447 ) );
      u[0] = Line.GetParameter( Line * s[0] );
      u[1] = Line.GetParameter( Line * s[1] );
      u[2] = Line.GetParameter( Line * s[2] );
   }
   else if ( Type == PointsOnConic2d )
   {
      Line.Def( P[0], conic.GetPolar( P[0] ).GetDir( ) );
      u[0] = 0;
      u[1] = Line.GetParameter( conic.GetPolar( P[1] ) * Line );
      u[2] = Line.GetParameter( StrL2d( P[1], P[2] ) * Line );
```

```
    }
    else
    {
        Line = s[0];
        u[0] = Line.GetParameter( conic.GetPole( s[0] ) );
        u[1] = Line.GetParameter( Line * s[1] );
        u[2] = Line.GetParameter( Line * s[2] );
    }
}
```

If the projective scale is of type *PencilOfLines2d*, we initialize Line with "weird" but sensible values that are not out of range. Then we intersect it with the three defining elements of the projective scale and store the corresponding parameter values.

We implemented some standard "getters," "setters," "markers," and "drawers." They work correctly only if the type of the projective scale is appropriate. Otherwise, they do nothing. It would be useless, for example, to draw the defining elements of a pencil of points. The most important task to be performed by *ProjScale2d* is the computation of a point or line to a given cross ratio. Consider, for example, the member function PointOfCrossRation(...):

Listing from "H/proj_geom.h":

```
P2d ProjScale2d::PointOfCrossRatio( const Real ratio )
{
    Real v;
    if ( fabs( ( u[1] - u[2] ) * ratio - u[0] + u[2] ) < 1e-6 )
        v = 1e12;
    else
        v = ( u[0] * ( u[1] - u[2] ) * ratio -
              ( u[0] - u[2] ) * u[1] ) /
              ( ( u[1] - u[2] ) * ratio - u[0] + u[2] );
    P2d V = Line.InBetweenPoint( v );
    if ( Type == PencilOfPoints2d )
        return V;
    else
    {
        int n;
        P2d W;
        StrL2d h( P[1], V );
        n = conic.SectionWithStraightLine( h, V, W );
        if ( n )
        {
            if ( fabs( h.GetParameter( V ) ) <
                 fabs( h.GetParameter( W ) ) )
```

```
            V = W;
         return V;
      }
      else
         return P[0];
   }
}
```

Here you see the advantage of storing the reals u[0], u[1], and u[2]. At the beginning of PointOfCrossRatio(...) we compute a real value v such that CR(u[0], u[1], u[2], v) is the input cross ratio. Then we distinguish two cases. If the scale is of type *PencilOfPoints2d*, we can immediately return the point V on Line that corresponds to v. Otherwise (if the scale is defined on the points of a conic), we have to project V onto the conic. That is, we have to invert the transformation of the projective scale onto Line (compare the listing of ComputeReals( )).

It is high time to give you an example of how all these classes and methods work. Later, we will go on to describing the remaining methods of importance.

Example 5.4. Projective scales
In "proj_scales.cpp" we use four projective scales SP (Scale of Points), SL (Scale of Lines), SCP (Scale of Conic Points), and SCL (Scale of Conic Lines) of different type. We define them in Init( ):

Listing from "proj_scales.cpp":

```
void Scene::Init( )
{
   StrL2d s( P2d( 0, -10 ), Xdir2d );
   P2d S[3];
   S[0] = s.InBetweenPoint( 0 );
   S[1] = s.InBetweenPoint( -5 );
   S[2] = s.InBetweenPoint( 10 );
   SP.Def( s, S );

   P2d P( 0, -2 );
   StrL2d p[3];
   p[0].Def( P, V2d( 0.8, 1 ) );
   p[1].Def( P, V2d( 0.0, 1 ) );
   p[2].Def( P, V2d( -0.8, 1 ) );
   SL.Def( P, p );

   Conic c;
   P2d C[5];
   C[0].Def( -5, 5 );
```

```
    C[1].Def( -14, 5 );
    C[2].Def( -10, 2 );
    C[3].Def( -10, 8 );
    C[4].Def( - 7, 3 );
    c.Def( Blue, 150, C );
    SCP.Def( c, C );

    StrL2d t[5];
    int i;
    for ( i = 0; i < 5; i++ )
    {
       t[i] = c.GetPolar( C[i] );
       t[i].Def( C[i], t[i].GetDir( ) );
       t[i].Reflect( Yaxis2d );
    }
    t[2].Translate( V2d( 0.5, -1 ) );
    D.Def( Red, 150, t );
    SCL.Def( D, t );

    R.Def( 2.5, -0.02, -3.5, 3.5, LINEAR );
}
```

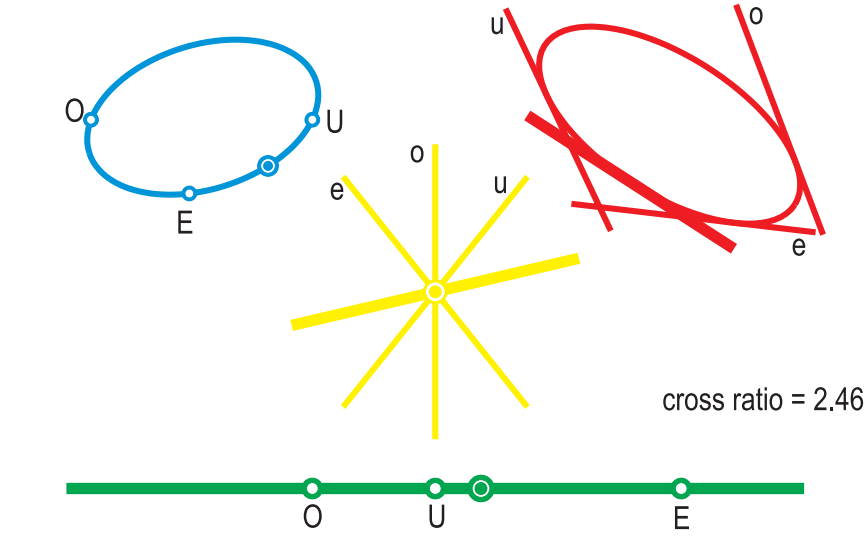

FIGURE 5.10. Diverse projective scales.

The variable D is of type *Conic*. We have to declare it globally because we need it in Draw( ) for the drawing of the conic tangents; R is a pulsing real. It will give the cross ratio of the fourth point/line to be displayed beside the defining

elements of the projective scales. Now we can draw and mark the projective scales. We start with the pencil of points. At the beginning of Draw( ) we write the following lines:

Listing from "proj_scales.cpp":

```
Real ratio = R.Next( );

SP.Draw( Green, −15, 15, VERY_THICK );
SP.MarkElements( Green, 0.35, 0.2 );
P2d Point;
Point = SP.PointOfCrossRatio( ratio );
Point.Mark( Green, 0.45, 0.3 );
Point.Mark( Green, 0.2 );
```

Similarly, we proceed with the pencil of lines and the points on the conic. We just have to take care to make the drawing of lines before the marking of points. The set of conic tangents needs a little trick. We compute the element of given cross ratio and immediately redefine it in order to draw a line segment symmetric to the point of tangency on the conic. It is here where we need the conic D already used in Init( ). In our example, the method GetPole(...) of the class *Conic* returns the point of tangency.

Listing from "proj_scales.cpp":

```
SCL.Draw( THICK );
SCL.DrawElements( Red, −5, 5, THICK );
StrL2d Line;
Line = SCL.LineOfCrossRatio( ratio );
Line.Def( D.GetPole( Line ), Line.GetDir( ) );
Line.Draw( Red, −5, 5, VERY_THICK );
```

Now you can start the program and run the animation. You will see four elements of the same cross ratio (Figure 5.10). As an additional feature we print this ratio on the screen. This allows you to verify that the defining elements of the projective scales correspond to the values 0, 1, and ∞, respectively. ♢

The class *ProjScale2d* on its own is quite useful. However, we designed it rather to be a member of the class *Projectivity2d*, which will be described later. For this purpose three more methods are of importance: The "=" operator and the two CrossRatio(...) methods. The latter take an input point or line and return the cross ratio defined by the three base elements of the projective scale. It is here where we make use of the fourth point, line, or real number of the private member variables:

Listing from "H/proj_geom.h":

```
Real ProjScale2d::CrossRatio( const P2d &A, const Real eps,
                              const Real infinity )
{
   Real ratio = 0;
   P[3] = A;
   if ( Type == PencilOfPoints )
   {
      if ( !SetCrossRatio( ratio, P, eps, infinity ) )
         ratio = infinity;
   }
   else if ( Type == PointsOnConic )
   {
      if ( !SetCrossRatio( ratio, conic, P, eps, infinity ) )
         ratio = infinity;
   }
   else
      Write( "warning:bad base type!" );
   return ratio;
}
```

Now we have everything to introduce a new class: *Projectivity2d*. Basically, it consists of two projective scales. Points that together with the defining elements of the scales define the same cross ratio correspond in a projective map. Additionally, we have private members to store the type of the scales and the upper and lower borders for estimation of 0 and ∞.

Listing from "H/proj_geom.h":

```
class Projectivity2d
{
public:
   Projectivity2d( );
   void Def( ProjScale2d &InScale1, ProjScale2d &InScale2,
         const Real eps = 1e-6, const Real infinity = 1e6 );
   P2d MapPointToPoint( const P2d &A );
   StrL2d MapPointToLine( const P2d &A );
   P2d MapLineToPoint( const StrL2d &a );
   StrL2d MapLineToLine( const StrL2d &a );
   P2d InvMapPointToPoint( const P2d &A );
   StrL2d InvMapPointToLine( const P2d &A );
   P2d InvMapLineToPoint( const StrL2d &a );
   StrL2d InvMapLineToLine( const StrL2d &a );
   void MarkElements( Color col, const Real rad1,
      const Real rad2 = 0 );
```

```
    void MarkElements( Color col[3], const Real rad1,
       const Real rad2 = 0 );
    void ConnectElements( Color col, ThinOrThick thick );
    void ConnectElements( Color col[3], ThinOrThick thick );
    P2d IntersectLines( Real ratio );
    StrL2d ConnectingLine( Real ratio );
    ProjType2d GetType1{ return Type1; }
    ProjType2d GetType2{ return Type2; }
  private:
    ProjScale2d Scale1, Scale2;
    ProjType2d Type1, Type2;
    Real eps, infinity;
  };
```

The class provides methods for transforming points and lines in both directions: from the first to the second scale and reversed. Depending on the type of the scales, you have to use the correct transformation method. Otherwise, the result is unpredictable (however, your system should not crash). Sometimes it is useful to mark/draw the defining elements or to connect/intersect corresponding points/lines. We provide methods for that as well.

Example 5.5. Projective map
An example of the use of the class *Projectivity2d* is `"proj_map1.cpp"`. We define a projectivity between the points of a straight line and a conic section. The straight lines that connect corresponding points envelop a curve of class three that is displayed as well. The global variables in use are as follows:

Listing from `"proj_map1.cpp"`:

```
ProjScale2d Scale1, Scale2;
Projectivity2d Proj;
Conic Conic1;
Color Col[3] = { Red, Green, Blue };
```

We refer to the conic Conic1 in both sections Init( ) and Draw( ). So we declare it as a global variable. It is now really easy to implement the envelope of corresponding lines. We just have to use the Open Geometry class *ClassCurve2d*:

Listing from "proj_map1.cpp":

```
class MyClassCurve: public ClassCurve2d
{
public:
   StrL2d Tangent( Real t )
   {
      return Proj.ConnectingLine( tan( t ) );
   }
};
MyClassCurve ClassCurve;
```

The tan(...) function in the argument of ConnectingLine(...) is necessary in order to transform a finite parameter interval of length π to $(-\infty, \infty)$. Additionally, it ensures a good distribution of curve tangents. The implementation of the projectivity is done in Init( ):

Listing from "proj_map1.cpp":

```
P2d P1[3];
P1[0].Def( -5, 6 );
P1[1].Def( 5, 6 );
P1[2].Def( 0, 12 );
Conic1.Def( Black, 150, P1, Origin, Xaxis2d );

P2d P2[5];
P2[0].Def( 15, 0 );
P2[1].Def( -15, 0 );
P2[2].Def( 0, 0 );

Scale1.Def( Conic1, P1 );
Scale2.Def( Xaxis2d, P2 );

Proj.Def( Scale1, Scale2 );
```

We define the conic section by three points and one point plus tangent. The three defining points serve for the definition of the projective scale as well. That is already everything as far as geometry is concerned. We have only to produce an attractive picture (Figure 5.11) and a little animation.

One global constant and three global variables are used for that:

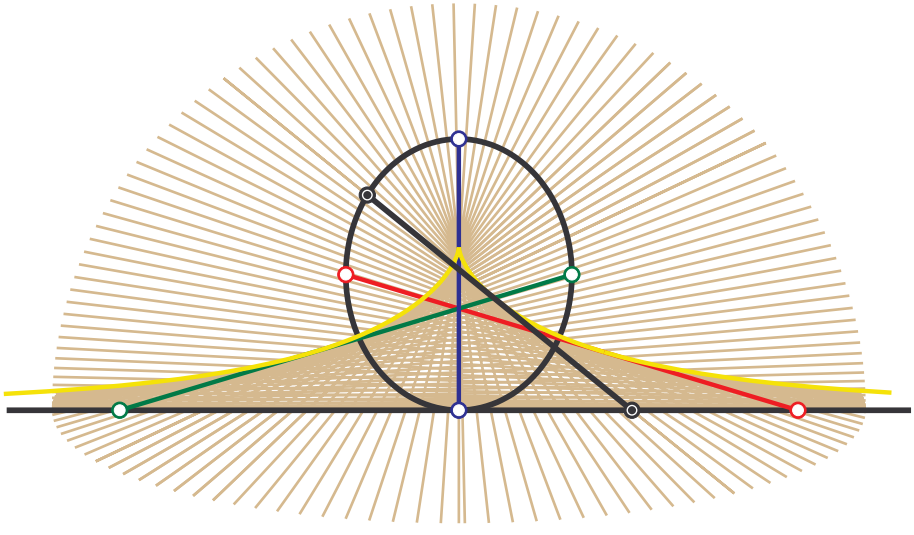

FIGURE 5.11. The connecting lines of a projectivity between a conic and a straight line envelop a curve of class three.

Listing from `"proj_map1.cpp"`:

```
const int N = 100;
StrL2d S[N+1];
int Count;
```

In Draw( ) we take a straight line through the conic center, rotate it about the conic center according to the frame number, and determine the two intersection points A and B. Then we transform the intersection point on the positive half ray and, while Count is within range, add the straight line AB to the array of lines. The rest of Draw( ) uses the convenient drawing and marking methods of *ProjScale2d* and *Projectivity2d*.

Listing from `"proj_map1.cpp"`:

```
void Scene::Draw( )
{
   P2d A, B;
   StrL2d s;
   P2d M = Conic1.GetM( );
   s.Def( M, V2d( 1, 0 ) );
```

```
    s.Rotate( M, 15 + 4 * FrameNum( ) );
    if ( Conic1.SectionWithStraightLine( s, A, B ) )
    {
       if ( Count < N )
          Count++;
       if ( s.GetParameter( A ) < s.GetParameter( B ) )
          A = B;
       B = Proj.MapPointToPoint( A );
       StraightLine2d( Black, A, B, THICK );
       S[Count].Def( A, B );
    }
    int i;
    for ( i = 0; i <= Count; i++ )
       S[i].Draw( LightYellow, -40, 40, THIN );

    Scale1.Draw( THICK );
    Scale2.Draw( Black, -20, 20, THICK );
    Proj.ConnectElements( Col, MEDIUM );
    ClassCurve.Draw( MEDIUM, 10 );
    Proj.MarkElements( Col, 0.3, 0.2 );
    A.Mark( Black, 0.3, 0.2 );
    A.Mark( Black, 0.15 );
    B.Mark( Black, 0.3, 0.2 );
    B.Mark( Black, 0.15 );
}
```

In a similar way we can use a projectivity between a pencil of lines and the tangent set of a conic to create a plane curve. The corresponding code does not differ essentially from `"proj_map1.cpp"` and can be found in `"proj_map2.cpp"`. ♢

The implementation of 3D projective scales (*ProjScale*) and 3D projectivities (*Projectivity*) is very similar to the two-dimensional case. Instead of giving detailed explanations, we refer the reader to the file `"proj_geom.h"` for further information. We will rather discuss another sample file (`"twisted_cubic.cpp"`).

Example 5.6. A twisted cubic
Let $a_i(\pi_i)$ be three projectively linked pencils of planes ($i = 1, 2, 3$). "Projectively linked" means in this context that projective scales are defined in each pencil. In general, planes $\pi_1(t)$, $\pi_2(t)$, and $\pi_3(t)$ of the same cross ratio t intersect in a common point $P(t)$. The locus $P(t)$ of these points is a space curve c. If $P(t)$ is defined for all cross ratios $t \in \overline{\mathbb{R}}$ (i.e., if the planes $\pi_i(t)$ have no common straight line), the generated space curve c is a *twisted cubic*.

It is easy to construct c with the help of the classes in `"proj_geom.h"`. We use three globally defined projective scales, three triangles that will visualize corresponding planes of the pencil, and a real variable T for the animation.

Listing from `"twisted_cubic.cpp"`:

```
ProjScale Scale1, Scale2, Scale3;
Poly3d Tri1, Tri2, Tri3;
Real T;
```

The twisted cubic can now be implemented as follows:

Listing from `"twisted_cubic.cpp"`:

```
class MyTwistedCubic: public ParamCurve3d
{
public:
    P3d CurvePoint( Real t )
    {
        Real TanT = tan( t );
        P3d P;
        Plane p[3];
        p[0] = Scale1.PlaneOfCrossRatio( TanT );
        p[1] = Scale2.PlaneOfCrossRatio( TanT );
        p[2] = Scale3.PlaneOfCrossRatio( TanT );
        if ( p[0].SectionWithTwoOtherPlanes( p[1], p[2], P ) )
            return P;
        else
            return P3d( 1e10, 1e10, 1e10 );
    }
};
MyTwistedCubic TwistedCubic;
```

Note that we transform the parameter once more with the tangent function in order to compensate the cross ratio's bad parameter distribution. Before computing the twisted cubic's points in Init( ), we must, of course, define the projective scales:

Listing from `"twisted_cubic.cpp"`:

```
const Real a = 5;

const P3d P1( a, 0, 0 );
const P3d P2( 0, 0, 0 );
const P3d P3( 0, a, a );

StrL3d axis1, axis2, axis3;
axis1.Def( P1, Zdir );
axis2.Def( P2, Ydir );
axis3.Def( P3, Xdir );

Plane plane1[3], plane2[3], plane3[3];

plane1[0].Def( P3d( 0,  0, 0 ), axis1 );
plane1[1].Def( P3d( 0,  a, 0 ), axis1 );
plane1[2].Def( P3d( 0, -a, 0 ), axis1 );

plane2[1].Def( P3d( a, 0,  0 ), axis2 );
plane2[0].Def( P3d( a, 0,  1 ), axis2 );
plane2[2].Def( P3d( a, 0, -1 ), axis2 );

plane3[0].Def( P3d( 0,  0, 0 ), axis3 );
plane3[2].Def( P3d( 0,  a, 0 ), axis3 );
plane3[1].Def( P3d( 0, -a, 0 ), axis3 );

Scale1.Def( axis1, plane1 );
Scale2.Def( axis2, plane2 );
Scale3.Def( axis3, plane3 );

const Real t1 = -0.5 * PI, t2 = -t1;
TwistedCubic.Def( Black, 150, t1, t2 );
```

The axes lie on three pairwise skew edges of a cube. The projective scales are declared in a way to produce a twisted cubic with only one real point at infinity (they are more attractive than their colleagues with three ideal points). Now, in order to display the curve's generation, we define the three triangles. Two vertices will always be incident with the axis of a pencil.

Listing from "twisted_cubic.cpp":

```
Tri1.Def( Red, 3 );
Tri2.Def( Green, 3 );
Tri3.Def( Blue, 3 );

Tri1[1] = P1;
Tri1[2].Def( a, 0, a );
Tri2[1] = P2;
Tri2[2].Def( 0, a, 0 );
Tri3[1] = P3;
Tri3[2].Def( a, a, a );
```

The common third vertex lies on the cubic. Hence, the triangles lie in planes that correspond in the projectivities between the three pencils of planes. The following lines stem from Draw( ). Varying the parameter T yields the animation.

Listing from "twisted_cubic.cpp":

```
Tri1[3] = TwistedCubic.CurvePoint( T );
Tri2[3] = Tri1[3];
Tri3[3] = Tri1[3];
Tri1[3].Mark( Black, 0.3, 0.2 );

Tri1.ShadeWithContour( DarkRed, MEDIUM );
Tri2.ShadeWithContour( DarkGreen, MEDIUM );
Tri3.ShadeWithContour( DarkBlue, MEDIUM );
```

♢

5.3 Open Geometry and Differential Geometry

Open Geometry puts special emphasis on differential geometric aspects of programming. Examples of this can be found throughout this book. In this section we compiled two sample programs where the differential geometric approach is evident.

Open Geometry is, of course, not a computer algebra system, i.e., you cannot perform symbolic calculus operations. You have to approximate the typical limiting processes of differential geometry by *discrete* models. In general, this is no problem and the loss of accuracy can be neglected.

Example 5.7. Osculating quadric
Let Φ be a ruled surface that is free of torsal generators. Any three pairwise skew generators g_0, g_1, and g_2 of Φ define a unique quadric $\Omega = \Omega(g_0, g_1, g_2)$ (the set of straight lines that intersect g_0, g_1, and g_2). If we perform the passage to the limit $g_1, g_2 \to g_0$, this quadric will converge to a well-defined limit Ω_0. It is called the *osculating quadric of* Φ *in* g_0 and plays an important role in the theory of ruled surfaces.

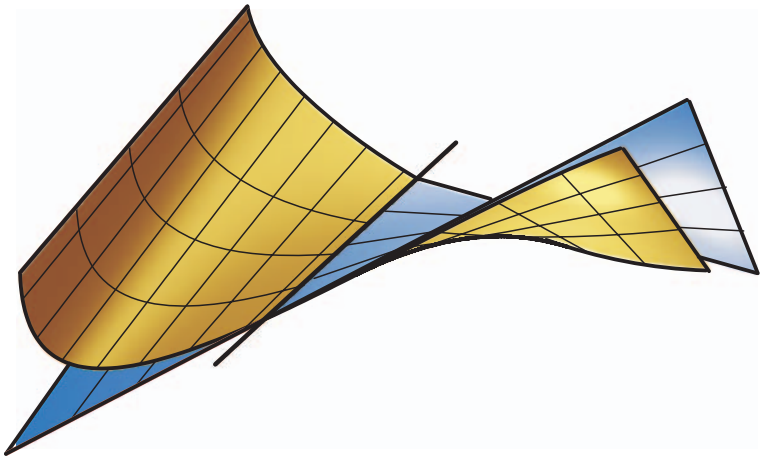

FIGURE 5.12. The osculating quadric of a ruled surface.

It is a very challenging and by far nontrivial task to produce a good and instructive picture that displays both, Φ and Ω at the same time. The reason for this is inherent in the problem. Because of the osculation along g_0, the z-buffer algorithm runs almost inevitably into problems.[3] These problems become even worse because in general, we will not be able to implement Ω_0 directly. We will have to use three "neighboring" generators for that purpose. Still, by using the right z-buffer settings (and one or another dirty trick) it is possible to overcome the difficulties.

The osculating quadric is defined by three neighboring generators. For convenient programming it is necessary to implement a successor class *Quadric* of *RuledSurface*. The template program for this class is `"quadric.cpp"`. It is rather short, so we will list it in one piece:

Listing of `"quadric.cpp"`:

```
#include "opengeom.h"
```

[3] Ω_0 lies on different sides of Φ in a neighborhood of g_0 and at the same time touches Φ along g_0.

```
#include "defaults3d.h"
#include "ruled_surface.h"

Quadric Q;

StrL3d r[3];
void Scene::Init( )
{
   r[0].Def( Origin, Xdir );
   r[1] = r[0];
   r[2] = r[0];
   r[0].Rotate( Zaxis, 30 );
   r[0].Translate( 0, 0, -5 );
   r[2].Rotate( Zaxis, -20 );
   r[2].Rotate( Yaxis, 10 );
   r[2].Translate( 0, 0, 5 );

   Q.Def( Yellow, 50, 50, -5, 5, 0, 1, r );
   Q.PrepareContour( );
}

void Scene::Draw( )
{
   Q.Shade( SMOOTH, REFLECTING );
   Q.DrawRulings( Black, 7, 7, THIN );
   Q.DrawBorderLines( Black, MEDIUM );
   Q.Contour( Black, MEDIUM );
}
```

You see that the use of this class is really very simple. However, you should be aware of a few facts:

- The instance Q of *Quadric* is defined in the same way as an ordinary parameterized surface. The additional input parameter r is an array of three straight lines.
- The u-parameter range ($[-5, 5]$ in our example) refers to the parameterization of r[0]. If you define r[0] by a point $\vec{p}$ and a direction vector v, this parameterization is given by $\vec{x}(t) = \vec{p} + t \cdot \vec{v}/|\vec{v}|$.
- The v-parameter range $[0, 1]$ fills the surface region between r[0] and r[2].
- The v-parameter lines are generators on Q. However, this is not true of the u-parameter lines. Therefore, the WireFrame(...) method of *ParamSurface* may yield unwanted results. If you want to display the surface generators, you should use the method DrawRulings(...).

After these preparations we can pass on to our original aim. In the program "osc_quadric.cpp" we display the osculating quadric of a ruled surface. Besides taking into account the visibility problems, we must carefully choose the ruled surface. Otherwise, the osculating quadric might have a weird shape. For this reason we decide to define the ruled surface Φ via two Bézier curves b_1 and b_2 (or Border1 and Border2 as they are called in "osc_quadric.cpp"). That is, we implement the ruled surface as follows:

Listing from "quadric.cpp":

```
class MyRuledSurface: public RuledSurface
{
public:
   virtual P3d DirectrixPoint( Real u )
   {
      return Border1.CurvePoint( u );
   }
   virtual V3d DirectionVector( Real u )
   {
      return V3d( Border1.CurvePoint( u ),
                  Border2.CurvePoint( u ) );
   }
};
MyRuledSurface RuledSurf;
```

Compared with a definition via parameterized equations, this approach permits easy experimenting possibilities and good design control. We change the shape of the ruled surface by simply changing the control points of the border lines.

Now to the osculating quadric Ω. Locally, the generator of osculation g_0 separates two regions $O_1, O_2 \in \Omega$ that lie on different sides of Φ. This causes numeric problems with the z-buffer algorithm. We can improve the visibility performance by applying little translations to O_1 and O_2. Therefore, we have to split Ω into two parts (v1 and v2 are parameter limits of Φ; U0 is a global constant to determine the generator of osculation):

Listing from "quadric.cpp":

```
const Real eps = 1e-8;
Real v;
StrL3d ruling[3];
int i;
for ( i = 0; i < 3; i++ )
{
   v = 0.5 * ( ( 2 - i ) * v1 + i * v2 );
   ruling[i].Def( RuledSurf.SurfacePoint( U0, v ),
```

```
                RuledSurf.OscQuadrDir( U0, v, eps ) );
}

const Real u11 = 0, u12 = 3;
const Real u21 = −5, u22 = 0;
OscQuad1.Def( Blue, 41, 41, u11, u12, 0, 1, ruling );
OscQuad2.Def( Blue, 41, 41, u21, u22, 0, 1, ruling );
OscQuad1.PrepareContour( );
OscQuad2.PrepareContour( );

V3d normal1 = RuledSurf.NormalVector( U0, 0.5 * ( v1 + v2 ) );
V3d normal2 = normal1;
normal1 *= 0.0004;
normal2 *= −0.03;
OscQuad1.Translate( normal1 );
OscQuad2.Translate( normal2 );
```

We first define three rulings r[0], r[1], and r[2] of Ω with the help of the OscQuadDir(...) method of the class *RuledSurface*. This method returns the direction of the generator of the osculating quadric (or, equivalently, the tangent of the asymptotic line) in a surface point $P = P(\mathsf{u}, \mathsf{v})$ by computing the straight line through P that intersects two neighboring generators $g(u - \varepsilon)$ and $g(u + \varepsilon)$ of the ruled surface.

In our case, the rulings r[0], r[1], and r[2] correspond to the v-values v1, (v1 + v2)/2, and v2, respectively. With their help we can define both parts O_1 and O_2 of Ω. Note that the setting of the respective u-parameter limits is a little delicate. You may need several tries to find a good choice.

Finally, we translate O_1 and O_2 in the direction of the surface normal in the midpoint of the generator of osculation. This seems to be reasonable, but depending on the special example, other directions may yield better results. The optimal length of the translation vector has to be found by a trial and error method as well. Note that all those parameters depend heavily not only on the ruled surface but also on the projection you use.

In Draw( ) we shade the two surfaces and draw their parameter lines. In doing so, we occasionally adapt the standard offset. In general, we do not recommend this,[4] but here it improves the visibility computations of straight lines.

[4] OPEN GEOMETRY algorithms are optimized for a default value of *STD_OFFSET*= 1e−3.

Listing from "quadric.cpp":

```
void Scene::Draw( )
{
   // shade osculating quadrics
   OscQuad1.Shade( SMOOTH, REFLECTING );
   OscQuad2.Shade( SMOOTH, REFLECTING );

   const Real eps = 1e-2;
   OscQuad1.DrawRulings( Black, M, N, THIN, eps );
   OscQuad2.DrawRulings( Black, M, N, THIN, eps );
   OscQuad1.DrawRulings( Black, 2, 2, MEDIUM, eps );
   OscQuad2.DrawRulings( Black, 2, 2, MEDIUM, eps );
   OscQuad1.Contour( Black, MEDIUM );
   OscQuad2.Contour( Black, MEDIUM );

   // draw the generator of second-order contact
   P3d P = RuledSurf.SurfacePoint( U0, RuledSurf.v1 );
   P3d Q = RuledSurf.SurfacePoint( U0, RuledSurf.v2 );
   StraightLine3d( Black, P, Q, MEDIUM, 1e-2 );

   // draw ruled surface
   SetOpacity( 0.7 );
   RuledSurf.Shade( SMOOTH, REFLECTING );
   RuledSurf.WireFrame( DarkBrown, 10, 13, THIN, 2 * 1e-4 );
   SetOpacity( 1 );
   RuledSurf.DrawBorderLines( DarkBrown, MEDIUM, true, true, 1e-4 );
   RuledSurf.Contour( DarkBrown, MEDIUM );

   StrL3d generator = RuledSurf.Generator( U0 );
   generator.Draw( Black, -1,
      RuledSurf.DirectionVector( U0 ).Length( ) + 1, THICK );
}
```

So far, you might be satisfied with the output. It is certainly good enough for a demonstration on your computer screen. However, in order to produce Figure 5.12 we used one more dirty trick. First, we drew all lines (parameter lines, border lines, contour outlines) and exported them in EPS format. Then we erased some lines to get a line graphic with correct visibility. Next, we displayed the ruled surface and the osculating quadrics without any lines, exported them separately to an image-processing program, erased parts of it, and put them together again. Finally, we placed the EPS line graphic over this picture and got a shaded surface with parameter lines and *absolutely correct* visibility. Of course, this method is quite cumbersome, but the high-quality output should compensate for that. ♢

Example 5.8. Focal surfaces of a line congruence
In projective, affine, or Euclidean 3-space one can distinguish three basic elements of geometry: points, planes, and straight lines. From the theoretical point of view of projective geometry, the sets of all points and of all planes are completely equivalent (principle of duality). That is, the planes of projective 3-space $\mathbb{P}_3$ can be regarded as the points of a second projective space $\overline{\mathbb{P}}_3$. Thus, every point set of $\mathbb{P}_3$ corresponds to a set of planes in $\overline{\mathbb{P}}_3$.

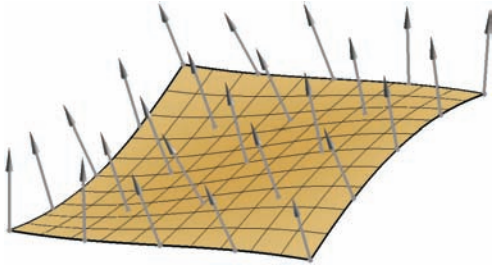

FIGURE 5.13. The director surface of a line congruence.

Straight lines, however, have no counterpart. They correspond to themselves and are called *self dual*. While the manifolds of all points or planes depend on three parameters, the manifold of all straight lines is of dimension four. Therefore, the theory of lines considers one-, two-, and three-parameter sets of straight lines. They are called *ruled surfaces*, *line congruences*, and *line complexes*, respectively ([27]).

We have already talked a great deal about ruled surfaces. In this place we will have a closer look at line congruences. Analytically, we can describe a line congruence $\mathcal{K}$ with the help of two functions $\vec{a}, \vec{e} : D \to \mathbb{R}^3$ that range in a domain $D \subset \mathbb{R}_2$. The point $(u_1, u_2) \in D$ corresponds to the straight line

$$x(u_1, u_2) \ldots \vec{x}(u_1, u_2, v) = \vec{a}(u_1, u_2) + \lambda \vec{e}(u_1, u_2), \qquad \lambda \in \mathbb{R}.$$

That is, $x(u_1, u_2)$ is the straight line with direction vector $\vec{e}(u_1, u_2)$ through $\vec{a}(u_1, u_2)$. The surface A described by $\vec{a}(u_1, u_2)$ is called the *director surface* of the congruence (Figure 5.13).

In the theory of line congruences, the *focal surfaces* play an important role ([27]). Geometrically speaking, the points of the focal surfaces are the cuspidal points of torsal generators on ruled surfaces contained in $\mathcal{K}$. Analytically, the focal points on a straight line $\vec{x}(u_1, u_2) \in \mathcal{K}$ are given by the solutions to the quadratic equation

$$\begin{aligned} v^2 \det(\vec{e}_{u_1}, \vec{e}_{u_2}, \vec{e}) + v[\det(\vec{e}_{u_1}, \vec{a}_{u_2}, \vec{e}) + \\ + \det(\vec{a}_{u_1}, \vec{e}_{u_2}, \vec{e})] + \det(\vec{a}_{u_1}, \vec{a}_{u_2}, \vec{e}) = 0. \end{aligned} \tag{3}$$

In general, that is, we get two solutions and therefore two (possibly complex) focal surfaces. With the help of formula (3) it is no problem to display the focal surfaces of a given congruence with OPEN GEOMETRY (`"line_congruence.cpp"`). First we need the vector functions $\vec{a}(u_1, u_2)$, $\vec{e}(u_1, u_2)$ and their derivatives:

Listing from `"line_congruence.cpp"`:

```
// director surface and derivates
const Real Eps = 1e-8, Eps2 = 0.5 / Eps;
const Real Factor = 0.3;
P3d A( Real u, Real u2 )
{
   return P3d( u, u2, Factor * u * u2 );
}
V3d Au1( Real u1, Real u2 )
{
   V3d au1 = A( u1 + Eps, u2 ) - A( u1 - Eps, u2 );
   au1 *= Eps2;
   return au1;
}
V3d Au2( Real u1, Real u2 )
{
   V3d au2 = A( u1, u2 + Eps ) - A( u1, u2 - Eps );
   au2 *= Eps2;
   return au2;
}

// direction vector and derivates
V3d E( Real u1, Real u2 )
{
   return V3d( Factor * u2, Factor * u1, -1 );
}
V3d Eu1( Real u1, Real u2 )
{
   V3d eu1 = E( u1 + Eps, u2 ) - E( u1 - Eps, u2 );
   eu1 *= Eps2;
   return eu1;
}
V3d Eu2( Real u1, Real u2 )
{
   V3d eu2 = E( u1, u2 + Eps ) - E( u1, u2 - Eps );
   eu2 *= Eps2;
   return eu2;
}
```

In our case the director surface is a hyperbolic paraboloid Δ. The lines of $\mathcal{K}$ meet Δ orthogonally (i.e., $\mathcal{K}$ is the *normal congruence of* Δ). Of course, the actual implementation of Δ is trivial. The SurfPoint(...) function just calls A(u, v). For the implementation of the focal surfaces we use two auxiliary functions. The first simply computes the determinant of three vectors of dimension three:

Listing from "line_congruence.cpp":

```
Real Determinant( V3d v1, V3d v2, V3d v3 )
{
   return v1.x * v2.y * v3.z + v1.y * v2.z * v3.x +
          v1.z * v2.x * v3.y - v1.z * v2.y * v3.x -
          v2.z * v3.y * v1.x - v3.z * v1.y * v2.x;
}
```

The second sets the parameter values of the focal points on the congruence line $x(u, v)$. For that purpose we have to solve a quadratic equation. The return value gives the number of real solutions.

Listing from "line_congruence.cpp":

```
int SetParamValues( const Real u, const Real v, Real &w1, Real &w2 )
{
   const Real a = Determinant( Eu1( u, v ), Eu2( u, v ), E( u, v ) );
   const Real b = Determinant( Eu1( u, v ), Eu2( u, v ), E( u, v ) )
      + Determinant( Au1( u, v ), Eu2( u, v ), E( u, v ) );
   const Real c = Determinant( Au1( u, v ), Au2( u, v ), E( u, v ) );
   return QuadrEquation( a, b, c, w1, w2, 1e-8 );
}
```

The actual implementation of the two focal surfaces reads as follows:

Listing from "line_congruence.cpp":

```
class FocalSurface: public ParamSurface
{
public:
   void Def( Color c, int m, int n, Real u1, Real u2,
             Real v1, Real v2, Boolean FirstSolution )
   {
      first_solution = FirstSolution;
      ParamSurface::Def( c, m, n, u1, u2, v1, v2 );
   }
```

```
    virtual P3d SurfacePoint( Real u, Real v )
    {
      Real w1 = 0, w2 = 0;
      if ( SetParamValues( u, v, w1, w2 ) )
      {
        if ( first_solution )
          return A( u, v ) + w1 * E( u, v );
        else
          return A( u, v ) + w2 * E( u, v );
      }
      else
        return Origin;
    }
  private:
    Boolean first_solution;
  };
  FocalSurface FocSurf1, FocSurf2;
```

Note the additional private member variable first_solution of type *Boolean*. It decides which solution to the quadratic equation is used to determine the focal point on $x(u, v)$. It allows us to implement both focal surfaces F_1 and F_2 in a unified way.

Additionally, we display the *midsurface* M of the congruence. Its points are the midpoints of the focal points on the congruence lines. Therefore, it can be implemented as follows:

Listing from `"line_congruence.cpp"`:

```
virtual P3d SurfacePoint( Real u, Real v )
{
  return 0.5 * ( FocSurf1.SurfacePoint( u, v ) +
                 FocSurf2.SurfacePoint( u, v ) );
}
```

In Init( ) we define the four surfaces; in Draw( ) we shade them. Additionally, we draw a couple of congruence lines in order to get the POV-Ray rendering displayed in Figure 5.14.

Listing from `"line_congruence.cpp"`:

```
const int m = 6, n = 6;
const Real u1 = DirSurf.u1, u2 = DirSurf.u2;
const Real v1 = DirSurf.v1, v2 = DirSurf.v2;
const Real du = ( u2 - u1 ) / ( m - 1.0 );
const Real dv = ( v2 - v1 ) / ( n - 1.0 );
Real u, v;
int i, j;
P3d D, F1, F2;
for ( i = 0, u = DirSurf.u1; i < m; i++, u += du )
for ( j = 0, v = DirSurf.v1; j < n; j++, v += dv )
{
   D = DirSurf.SurfacePoint( u, v );
   F1 = FocSurf1.SurfacePoint( u, v );
   F2 = FocSurf2.SurfacePoint( u, v );
   StraightLine3d( Black, D, F1, THIN );
   StraightLine3d( Black, D, F2, THIN );
   D.Mark( DarkBlue, 0.1 );
   F1.Mark( DarkRed, 0.1 );
   F2.Mark( DarkRed, 0.1 );
}
```

♢

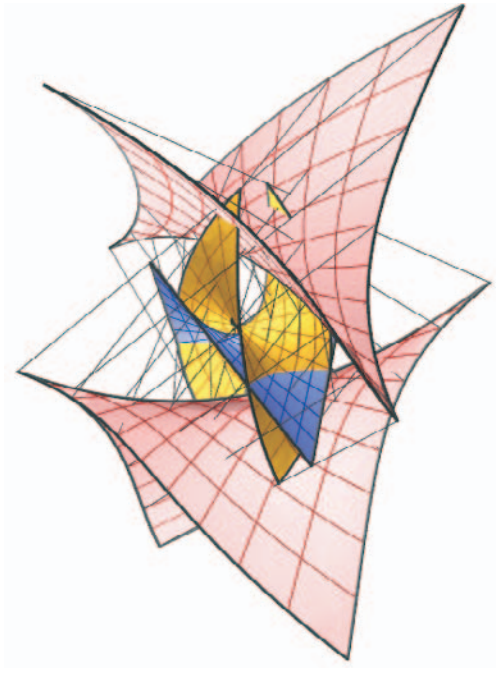

FIGURE 5.14. Director surface, focal surfaces, and midsurface of a line congruence.

6

Compendium of Version 2.0

In the following, we give a comprehensive listing of many of OPEN GEOMETRY's classes. Originally, we intended to list all of them, but this turned out to be impossible because of their sheer number. Many classes have been written for rather special internal tasks in OPEN GEOMETRY and are of little interest to the reader. So we picked out those classes that we believe to be most useful. We display them together with their relevant operators and methods.

If the code is not self-explanatory, we give a short description. Frequently, we provide a picture and/or few lines of sample code that show how to use the class. If the information contained in this chapter is not enough for you, you can find the corresponding headers in OPEN GEOMETRY's `"H/"`-directory. For each class the corresponding header file is indicated.

Note that the listings of this chapter are not exact copies of the source code. Occasionally, we dropped a line, rearranged the methods, or inserted additional line-breaks or comments. We did this for the sake of simplified reading and not to obscure things. You are still free to read the original and unabridged source code in the files of the `"H/"`-directory.

6.1 Useful 2D Classes and Their Methods

class *Arc2d*; → declaration in `"arc2d.h"`

Arc2d describes a segment of a 2D circle. It is a successor of *Sector2d* and has only one additional method:

void Draw(*ThinOrThick* style);
//This method calls the drawing method of *Sector2d*
// *without* the option of drawing the line segments that
// connect the arc center with start and end points, respectively.

class *Arrow2d*; → declaration in `"arrows2d.h"`

Arrow2d is a new OPEN GEOMETRY class for displaying different types of arrows in two dimensions. It is defined via start and end points. Additionally, the user can specify its dimensions and choose between single- and double- pointed arrows (compare the sample code below and Figure 6.1). The class is derived from *ComplexPoly2d* and inherits all drawing and shading methods. Consider also the class *BentArrow2d*.

Definition:

void Def(*Color* col,
Boolean left_right_arrow = false,
const *P2d* &first_point = *P2d*(−1, 0),
const *P2d* &second_point = *P2d*(0, 0),
Real width = 0.2,
Real relative_arrow_length = 0.3,
Real arrow_slope = 0.4);

Frequently used methods and operators:

void operator = (**const** *Arrow2d* &other);

Instead of using the class *Arrow2d*, the user may as well use the function DrawArrow2d(...) (compare Figure 6.1). Its header can be found in `"arrow2d.h"` as well:

void DrawArrow2d(*Color* col,
const *P2d* &P, **const** *P2d* &Q,
Real arrow_size = −1,
Real arrow_slope = 0.2,
FilledOrNot fill = *EMPTY*,
ThinOrThick style = *THIN*,
Boolean two_arrows = false,
int type = 1,
ComplexPoly2d *cp = NULL,
Real width = 0.5);

Sample code for better understanding:

```
Arrow2d Arr1, Arr2, Arr3;
Arr1.Def( Black, true );
Arr1.Scale( 6, 8 );
Arr1.Rotate( Origin2d, 45 );
Arr1.Translate( V2d( 0, 2 ) );
P2d p( 3, 2 ), q( 8, 2 );
Arr2.Def( Blue, false, p, q, 0.25, 0.5, 0.3 );
Arr3 = Arr2;
Arr3.Rotate( Origin2d, -20 );
DrawArrow2d( Black, P2d( -2, 2 ), P2d( -5, 5 ) );
Arr1.Shade( );
Arr1.Outline( Black, MEDIUM );
Arr2.Shade( );
Arr3.Outline( Red, THIN );
```

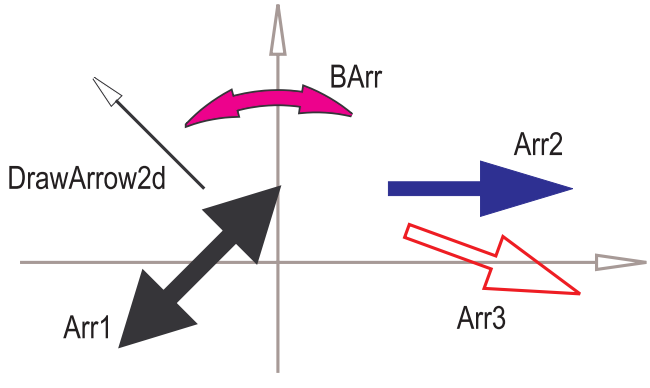

FIGURE 6.1. Different arrows in 2D.

class *BentArrow2d*; → declaration in `"arrows2d.h"`

Class for displaying curved arrows. As with *Arrow2d*, the user can choose between double- and single-pointed arrows and can specify the arrow dimension. The class is derived from *ComplexPoly2d* and inherits all drawing and shading methods. Compare also Figure 6.1.

Definition:

```
void Def( Color col,
          Boolean left_right_arrow,
          const P2d &P, const P2d &Q,
          Real radius, // signed
```

```
                    Real width = 0.5,
                    Real relative_arrow_length = 0.3,
                    Real arrow_slope = 0.25 );
```

Frequently used methods and operators:

```
void operator = ( const BentArrow2d &other );
```

Sample code for better understanding:

```
BentArrow2d BArr;
P2d P( 4, -2), Q = P;
Q.Rotate( Origin2d, 60 );
BArr.Def( Magenta, true, P, Q, P.Length( ) );
BArr.Rotate( Origin2d, 90 );
BArr.Outline( Black, MEDIUM );
BArr.Shade( );
```

class *BezierCurve2d*; → declaration in `"bezier.h"`

Describes an integral Bézier curve in 2D by its control polygon. The curve points are computed by means of DeCasteljau's algorithm. We provide methods for manipulating the curve (changing of control points, degree elevation, splitting). The base class is *ParamCurve2d*. Some methods are overwritten. For further information, please consult Section 2.6. Have a look at page 126 in Chapter 2 for more detailed information.

Constructors:

```
BezierCurve2d( ) { B = C = NULL; }
// B and C are dynamically allocated arrays of control points.
// C is needed for DeCasteljau's algorithm.
```

Definition:

```
void Def( Color c, int total_size,
          int size_of_control_poly, P2d Control[] );
void Def( Color c, int total_size,
          int size_of_control_poly, Coord2dArray Control );
// total_size is the number of curve points that will
// be computed, while size_of_control_poly is the number
// of control points to be used.
```

Frequently used methods and operators:

```
virtual P2d CurvePoint( Real u );
   // Calculate curve point by means of the DeCasteljau algorithm.
virtual StrL2d Tangent( Real u )/* const = 0 */;
void DrawControlPolygon( Color col, ThinOrThick thickness );
```

```
void MarkControlPoints( Color col, Real rad1, Real rad0 );
void SetControlPoint( int i, P2d P );
   // Redefine control point with index i
P2d GetControlPoint( int i ) { return B[i]; };
   // Return control point with index i
int GetPNum( ) { return PNum; }
   // PNum is the number of control points
void ElevateDegree( int i ); // Elevate the degree by i.
void ReversePolygon( void );
   // Reverse the order of the control points
   // (useful after having split the curve)
void Split( Real u, Boolean second_half );
   // Split the Bézier curve at the parameter value u. Depending
   // on the value of second_half, the new curve is identical to the
   // part of the old curve that corresponds to the parameter interval
   // [0, u] or [u, 1].
```

Sample code for better understanding:

```
BezierCurve2d BezierCurve;
Coord2dArray BasePoints = { {−10, −7}, {−8, 4},
   {5, 7}, {10, −7} };
BezierCurve.Def( Black, 200, 4, BasePoints );
BezierCurve.ElevateDegree( 2 );
BezierCurve.Split( 0.5 );
BezierCurve.Draw( VERY_THICK );
BezierCurve.DrawControlPolygon( Blue, THICK );
BezierCurve.MarkControlPoints( Blue, 0.15, 0.1 );
```

class *Circ2d*; → declaration in `"circ2d.h"`

Derived from *O2d*, this class describes a circle in 2D. Many new methods have been added in Open Geometry 2.0. The circle is approximated by a regular polygon. The number of vertices depends on the circle radius. Sometimes it is advisable to use an instance of the class *RegPoly2d* with a sufficiently high number of vertices instead of a circle (e.g., when you want to shade the object).

Constructors:

```
Circ2d( ) // Default constructor.
Circ2d( Color f, const P2d &Mid, Real r,
   FilledOrNot filled = EMPTY ) // Classic definition.
Circ2d( Color f, const P2d &A, const P2d &B, const P2d &C,
   FilledOrNot filled = EMPTY ) // Definition by three points.
```

Definition:

```
void Def( Color f, const P2d &M, Real r,
    FilledOrNot filled = EMPTY ) // Center plus radius.
void Def( Color f, Real xm, Real ym, Real r,
    FilledOrNot filled = EMPTY ); // Center coordinates plus radius.
void Def( Color f, const P2d &A, const P2d &B, const P2d &C,
    FilledOrNot filled = EMPTY ); // Three points.
```

Additional methods:

```
P2d Mid( ) const  // Return the center.
P2d GetCenter( ) const // Return the center.
Real GetRadius( ) const // Return the radius.

StrL2d GetTangent( const P2d &T ) const;
    // Return the polar of T with respect to the circle
    // and give a warning if T does not lie on the circle.
StrL2d GetNormal( const P2d &P ) const;
    // Return the circle normal through P.
V2d GetTangentVector( const P2d &T ) const;
   // Return the normalized direction of the polar of T with respect
   // to the circle and give a warning if T does not lie on the circle.
V2d GetNormalVector( const P2d &P ) const;
   // Return the normalized vector in direction of MP (with arrow), where M is
   // the circle center

// Methods to intersect circle with the straight line g. The number of
// real intersection points is either stored in n or returned. The real
// intersection points are stored in S1 and S2.
void SectionWithStraightLine( const StrL2d &g, P23d &S1,
                                              P23d &S2, int &n ) const;
int SectionWithStraightLine( const StrL2d &g, P23d &S1,
                                              P23d &S2 ) const;
// Intersection of two circles. Arguments have meaning as in
// preceding method.
int SectionWithCircle( const Circ2d &k, P23d &S1,
                                          P23d &S2 ) const;

void InvertPoint( P2d &P ) const; // Invert point at circle.
P2d GetPole( const StrL2d &Polar ) const; // Return pole.
StrL2d GetPolar( const P2d &Pole ) const; // Return polar.

// Unique drawing method.
void Draw( ThinOrThick thick = THIN ) const;
```

Sample code for better understanding:

```
Circ2d circle;
circle.Def( Black, P2d( -5, 2 ), 5 );
StrL2d s( P2d( -3, 0 ), V2d( 0.5, 2 ) );
P2d S1, S2;
circle.SectionWithStraightLine( s, S1, S2 );
StrL2d tangent1, tangent2;
tangent1 = circle.GetTangent( S1 );
tangent2 = circle.GetTangent( S2 );
P2d Pole = tangent1 * tangent2;
StrL2d polar = circle.GetPolar( Pole );
if ( !( polar == s ) )
   ShowString( "Something is wrong!" );
circle.Draw( THICK );
StraightLine2d( Red, S1, S2, MEDIUM );
StraightLine2d( Black, Pole, S1, MEDIUM );
StraightLine2d( Black, Pole, S2, MEDIUM );
S1.Mark( Black, 0.2, 0.1 );
S2.Mark( Black, 0.2, 0.1 );
Pole.Mark( Red, 0.2, 0.1 );
```

class *ClassCurve2d*; → declaration in `"lines2d.h"`

Describes a plane curve via its tangents rather than via its points. In order to use this class you have to implement the virtual member function CurvePoint(...) (similar to the class *ParamCurve2d*).

Member functions:

```
virtual StrL2d Tangent( Real u ) = 0;
virtual P2d CurvePoint( Real u );
virtual V2d TangentVector( Real u );
```

class *ComplexL3d*; → declaration in `"lines3d.h"`

Describes a 3D line that, in contrast to *L3d*, may consist of different branches. The intersection of two surfaces or polyhedra will, e.g., be stored as *ComplexL3d* object. Base class.

Constructors:

ComplexL3d( )// Default constructor.

Definition:

```
void Def( Color f, int number_of_branches, int size[ ] );
   // number_of_branches curves (objects of type L3d) of size[i]
```

```
    // points each.
```

Operators and methods:

```
L3d & operator [ ] ( int i ) const. // Return the i^th curve.
friend O3d & operator * ( ComplexL3d &w, Plane &e );
    // Intersect complex line with plane.

void Draw( Color f, ThinOrThick thick,
           Real offset = STD_OFFSET );
    // Standard drawing routine.
void MarkPoints( Color f, Real r1, Real r2 );
    // Mark all points with two small circles of radii r1 and r2.

int PointNum( ); // Return the number of points.
void AssignColor( Color f );

// Some geometric transformations.
void Translate( Real dx, Real dy, Real dz );
void Translate( const V3d &v );
void Scale( Real kx, Real ky, Real kz );
void Rotate( const StrL3d &a, Real w );
void Copy( ComplexL3d &result );

// Intersection methods.
void SectionWith( ComplexL3d &w, O3d &S );
void SectionWithPlane( Plane &e, O3d &S );
```

class *ComplexPoly2d*; → declaration in `"complex_poly.h"`

Closed, non convex polygon with holes (loops) in 2-space. Derived from *Poly2d*.

Constructors:

```
ComplexPoly2d( ); // Default constructor.
```

Definition:

```
void Def( Color f, int NLoops, Poly2d *loop, int oriented = 0 );
    // NLoops is the number of loops; loop is an array of polygons,
    // i.e., a pointer to the first polygon.
void Def( const ComplexPoly2d &cp ); // Copy complex polygon cp.
```

Additional methods:

```
void Translate( const V2d &t );
void Rotate( const P2d &center, Real angle_in_deg );
void Scale( Real kx, Real ky = DUMMY );

void Fill( );
void Shade( ); // Does not work in case of intersecting loops!
void Outline( Color col, ThinOrThick style ); // Draw the outline.
```

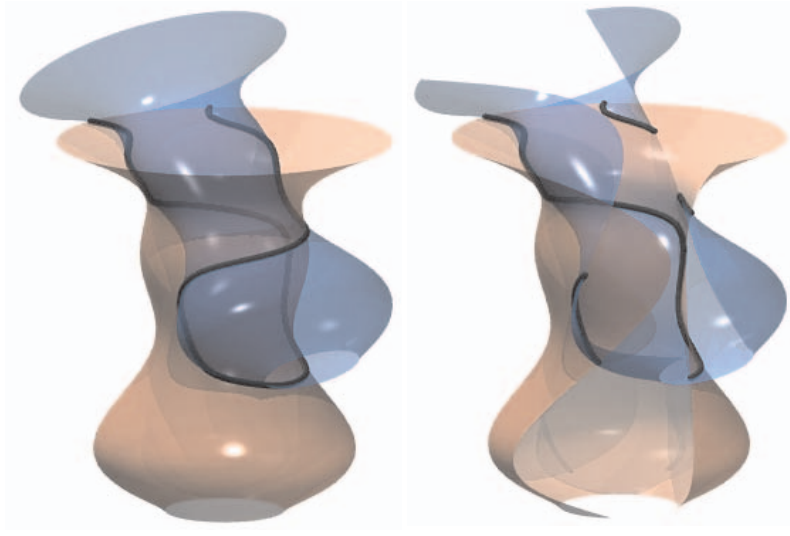

FIGURE 6.2. A complex line as intersection curve of two parameterized surfaces. It was calculated by means of the SectionWithSurface(...) method of *ParamSurface*. The algorithm works even if the intersection curve consists of several branches (right). All branches are objects of type *L3d*.

Sample code for better understanding:

```
// The complex polygon will consist of four loops (polygons).
const int nLoops = 4;
Poly2d p[nLoops];
int n = 40;
p[0].Def( Gray, n, FILLED );
Real delta = 2 * PI / n, fi = 0;
for ( int i = 1; i <= n; i++, fi += delta )
    p[0][i]( 2.5 * cos( fi ) -2.5, 3 * sin( fi ) );
p[1] = p[0]; p[1].Scale( 0.9, 0.6 ); p[1].Translate( -0.2, 0 );
p[2] = p[0]; p[2].Reflect( Yaxis2d ); p[2].Scale( 1.2 );
p[3] = p[1]; p[3].Reflect( Yaxis2d ); p[3].Scale( 1.2 );

// Now combine the four loops to one complex polygon.
ComplexPoly2d complex_poly;
complex_poly.Def( Gray, nLoops, p );
complex_poly.Shade( );
complex_poly.Outline( Black, MEDIUM );
```

class *Conic*; → declaration in `"conic.h"`

Describes a conic in 2-space or in 3-space in the $[x, y]$-plane. Base class.

Constructors:

```
Conic( ); // Default constructor.
```

Constructors:

```
Conic & operator = ( Conic &other );
```

Definition:

```
Conic & operator = ( Conic &other );
void Def( Color col, int numPoints, const P2d P[5] );
// numPoints is the number of points on the approximating
// polygon. Calculations with the conic are not done with these
// approximations but with the exact mathematical equation!
// The conic is defined by five points.
void Def( Color col, int numPoints, const P2d &P,
        const P2d &F1, const P2d &F2, TypeOfConic type );
// Point plus two focal points and type of conic;
// type can be either ELLIPSE or HYPERBOLA.
void Def( Color col, int numPoints, const P2d &P,
        const P2d &Q, const P2d &R, const P2d &F );
// Definition through three points plus focal point.
void Def( Color col, int numPoints, const StrL2d t[5] );
// Definition through five tangents.
void Def( Color col, int numPoints, const P2d P[3],
        const P2d &T, const StrL2d &t );
// Definition through three points and point plus tangent.
void Def( Color col, int numPoints, const P2d &M,
        const P2d &A, const P2d &B );
// Definition through pair of conjugate diameters; possibly
// points of the axes.
void Def( Color col, int numPoints, Real d[6] );
// Definition through coefficients of implicit equation.
```

Frequently used methods and operators:

```
void Draw( ThinOrThick thick, Real max_radius = DUMMY );
TypeOfConic type( ) const;
   // Return the type: ELLIPSE, HYPERBOLA,
   // PARABOLA, IRREGULAR.
P2d GetCenter( ) { return M; } // Return the center.
P2d GetA( ) const { return A; }
P2d GetB( ) const { return B; }
   // A and B are points on the major axis.
P2d GetC( ) const { return C; }
P2d GetD( ) const { return D; }
   // C and D are points on the minor axis.
P2d GetM( ) const { return M; } // Returns the center.
```

```
Real DistMA( ) const { return MA; }
Real DistMC( ) const { return MC; }
StrL2d MajorAxis( ) const { return Axis1; }
StrL2d MinorAxis( ) const { return Axis2; }
StrL2d Asymptote1( ) const { return As1; }
StrL2d Asymptote2( ) const { return As2; }
void GetCoefficients( Real c[6] ) const;
   // The equation of the conic is then
   // c[0]x^2 + c[1]xy + c[2]y^2 + c[3]x + c[4]y + c[5] = 0.
int SectionWithStraightLine( const StrL2d s, P2d &S1, P2d &S2 );
   // Return the number of intersection points (≤ 2)
   // and calculates the corresponding points S1 and S2.
void ChangeColor( Color c );
Color GetColor( ) { return col; }
int Size( ) { return size; }
int SectionWithConic( const Conic &c, P2d S[4] ) const;
   // Return the number of intersection points (≤ 4)
   // and calculates the corresponding points.
void WriteImplicitEquation( ) const;
   // Display the implicit equation
   // in a window on the screen.
StrL2d GetPolar( const P2d &Pole ) const;
   // Return polar of point.
P2d GetPole( const StrL2d &polar ) const;
   // Return pole of straight line.
Real EvaluateImplicitEquation( const P23d &P ) const;
Boolean IsOnConic( const P23d &P, Real tolerance ) const;
Boolean IsTangent( const StrL2d &s ) const;
P2d OscCenter( const P2d &P ) const;
   // Return center of osculations in conic point.
```

Sample code for better understanding:

```
#include "opengeom.h"
#include "defaults2d.h"

Conic Circ, Hyp;

void Scene::Init( )
{
   Real d[6]; // coefficients of implicit equation
   d[0] = 1, d[1] = 0, d[2] = 1,
   d[3] = 0, d[4] = 0, d[5] = -25;
   Circ.Def( Black, 101, d ); // circle of radius 5

   StrL2d t[6]; // six conic tangents
   t[0].Def( P2d( -1, -1 ), V2d( 0, 1 ) );
   t[1].Def( P2d( -1, 2 ), V2d( -0.5, 1 ) );
```

```
    t[2].Def( P2d( -3, -3 ), V2d( 0.5, 1 ) );
    t[3].Def( P2d( 3, -1 ), V2d( 0, 1 ) );
    t[4].Def( P2d( 5, 3 ), V2d( 1, 1 ) );
    Hyp.Def( Black, 101, t );
}
void Scene::Draw( )
{
    ShowAxes2d( Black, -10, 10, -8, 8 );
    Circ.Draw( THICK );
    Hyp.Draw( THICK );

    P2d S[4];
    int i, n;
    n = Circ.SectionWithConic( Hyp, S );
    for ( i = 0; i < n; i++ )
        S[i].Mark( Black, 0.2, 0.1 );
}
```

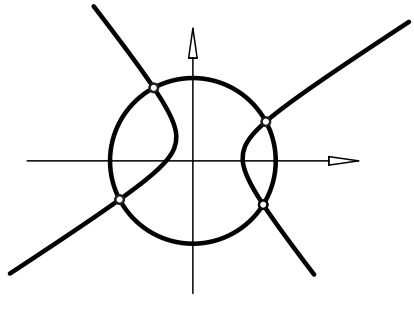

FIGURE 6.3. Two conics and their points of intersection (compare the above sample code).

class *DiffEquation*; → declaration in `"diff_equation.h"`

Describes a differential equation (2D vector field) and provides a solving method (RUNGE–KUTTA). Abstract class due to the purely virtual member function TangentVector(...). In order to use it, you must derive a class of your own from *DiffEquation*. It is derived from *L2d*. Thus, you can immediately plot the solution curve. Requires the inclusion of `"diff_equation.h"` at the beginning of your file.

Constructor:

DiffEquation( ); // Default constructor.

Additional methods:

virtual void Solve(*Real* t1, *Real* t2, *Real* h, **const** *P2d* &StartPoint);

```
    // Solves differential equation by means of RUNGE−KUTTA.
    //
  virtual V2d TangentVector( Real t, const P2d &P) = 0;
    // Virtual function to describe the corresponding 2D vector field.
    // Has to be implemented by the user!
```

Sample code for better understanding:

```
#include "opengeom.h"
#include "defaults2d.h"
#include "diff_equation.h"

class MyDiffEquation: public DiffEquation
{
  virtual V2d TangentVector( Real t, const P2d &P )
  {
    V2d v;
    v.Def( 0 * P.x + sin( 0.5 * t ), 3 * cos( 1.5 * t ) );
    v *= 3.5;
    return v;
  }
};
MyDiffEquation Curve;

void Scene::Init( )
{
  Curve.Solve( −6.3, 6.3, 0.03, P2d( −1, 1 ) );
  Curve.ChangeColor( Blue );
}
void Scene::Draw( )
{
  Curve.Draw( MEDIUM );
}
```

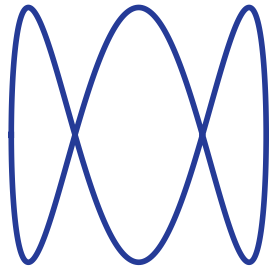

FIGURE 6.4. Integral curve of the differential equation from the above sample code.

class *FourBarLinkage*; → declaration in `"kinemat.h"`

Class to describe a four-bar linkage. It is equipped with methods for computing and displaying special positions (compare Figure 6.5 and [14], Chapter 9). Requires the inclusion of `"kinemat.h"` at the beginning of the program.

Constructors:

FourBarLinkage( ) // Default constructor.

Definition:

void Def(*Real* LA, *Real* AB, *Real* MB, *Real* LM,
Real start_alpha_in_deg, *Real* delta_alpha);
// Define mechanims by length of the four bars and starting angle
// (compare Figure 6.5). delta_alpha is the angle increment for the
// animation.
void Def(**const** *P2d* &L, **const** *P2d* &A, **const** *P2d* &B,
const *P2d* &M, *Real* delta_alpha = 1);
// Define mechanims by position of joints (Figure 6.5).
// delta_alpha is the angle increment for the animation.

Methods:

void Draw(*Color* c, *ThinOrThick* thick,
Real r1, *Real* r2, *Boolean* with_names,
Boolean show_circles);
// Draw mechanism and marks relevant points with radii r1 and r2.
// If show_circles is **true**, the path circles of A and B will be
// displayed.
P2d CalcPoint(*Real* AC, *Real* BC, **int** side = 1,
Boolean calc_N = false);
// Compute the position of a point C that is attached to the
// mechanism (compare Figure 6.5). C will be stored
// as protected member variable of the class.
void DrawTriangle(*Color* c, *Real* r1, *Real* r2);
// Draw the triangle that connects C with the mechanism

```
    // and marks its vertices (compare CalcPoint).
  void Move( ); // Animate the mechanism.
  void PutAtPosition( Real x, Real y, Real rot_angle_in_deg );
    // Translate mechanism by (x, y) and rotate it about
    // the new position of L.
  void CalcAlternateMechanisms( FourBarLinkage &Alt1,
                                FourBarLinkage &Alt2 );
    // Compute the two alternate mechanisms according to ROBERTS.

  // Methods to return important points and lengths (Figure 6.5):
  P2d GetL( );
  P2d GetM( );
  P2d GetA( );
  P2d GetB( );
  Real GetLM( );
  Real GetAB( );
  Real GetLA( );
  Real GetLB( );
  Real Get_alpha_in_deg( ); // Return current angle.
  Real GetDeltaAlpha( ); // Return current angle increment.
  void ChangeDeltaAlpha( Real k ); // Change current angle.

  P2d GetPole( ); // Get instantanious pole of the motion.
  StrL2d GetPolodeTangent( ); // Polode tangent in instantaneous pole.
  P2d CenterOfOsculatingCircle( const P2d &C );
    // Center of osculation for arbitrary
    // point connected with mechanism.
  Boolean PathClosed( ); // Check whether path is already closed.
```

Sample code for better understanding:

```
#include "opengeom.h"
#include "kinemat.h"

FourBarLinkage Mechanism;
PathCurve2d PathC( PureRed );

void Scene::Init( )
{
  Mechanism.Def( 7, 11, 12, 15.7, 0, -1 );
  Real rot_angle = 90;
  V2d trans_vector( 0, 0 );
  Mechanism.PutAtPosition( trans_vector.x, trans_vector.y,  rot_angle );
  ScaleLetters( 2 );
}
void Scene::Draw( )
{
  const Real r1 = 0.2, r2 = r1 / 2;
```

```
    P2d C = Mechanism.CalcPoint( 8, 9 );
    if ( !Mechanism.PathClosed( ) )
      PathC.AddPoint( C );
    Mechanism.DrawTriangle( Green, 2 * r1, r1 );
    Mechanism.Draw( Black, VERY_THICK, r1, r2, true, true );
    PathC.Draw( THIN );
    PathC.MarkPoints( Pink, 0.2, 0, 10 );
    C.AttachString( Black, -3 * r1, - 5 *r1, "C" );
}
void Scene::Animate( )
{
    Mechanism.Move( );
}
void Scene::CleanUp( )
{
}
void Projection::Def( )
{
    if ( VeryFirstTime( ) )
    {
      xyCoordinates( -12.0, 12.0, -10.0, 10.0 );
    }
}
```

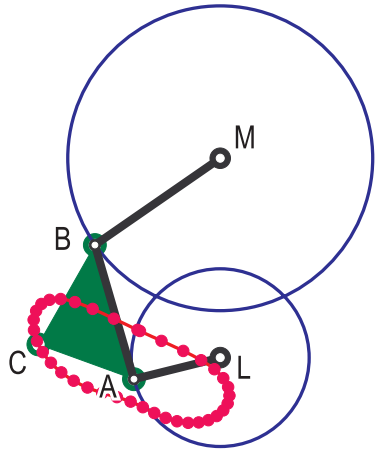

FIGURE 6.5. A four-bar linkage.

class *Function*; → declaration in `"function2d.h"`

Describes a 2D function graph and provides methods for drawing, finding zeros and extremal values, numerical computation of first and second derivatives, numerical integration, etc. Requires the inclusion of

`"function2d.h"` at the top of the file. Base class. In Section 7.4 you can see how to add a new method GetMinMax(...) to this class.

Constructors:

```
Function( void ); // Default constructor.
Function( Real (*function)( Real ) )
   // Needs pointer to a function as argument.
```

Definition:

```
void Def( Real (*function)( Real ) ); // See constructor.
```

Methods:

```
void Draw( Color col, Real x1, Real x2, int n,
                ThinOrThick style, Real max_dist = -1 );
   // Draw the function graph between x1 and x2. n is the number
   // of points on the approximating polygon; style determines the
   // line style to be used. Two points will be connected only if their
   // distance is less than max_dist.

// Methods for computing the zeros:
int CalcZeros( Real xmin, Real xmax, int n = 100,
                Real tolerance = 1e-8 );
   // Efficient method to compute a maximum of n zeros in
   // [xmin, xmax] with the help of binary search. The return value
   // is the number of zeros. They are accessible via Solution[i].
Real ZeroWithNewton( Real x, Real tolerance );
   // Compute one zero with Newton iteration.
Real ZeroWithBinarySearch( Real x1, Real x2, Real tolerance );
   // Compute one zero with binary search.
Real Solution( int i ); // Access the zeros found with CalcZeros(...).
void MarkZeros( Color col, Real r1, Real r2 );
   // Mark the zeros found with CalcZeros(...).

Real operator ( ) ( Real x ); // Get return value of x.
P2d operator [ ] ( Real x ); // Get point (x, y) on function graph.
Real FirstDerivative( Real x ); // Return first derivate at x.
Real SecondDerivative( Real x ); // Return second derivate at x.
V2d VelocityVector( Real x ); // Return vector (1, f'(x)).
V2d AccelerationVector( Real x ); // Return vector (1, f''(x)).
void TransformIntoL2d( Color col, Real x1, Real x2, int n, L2d &line );
   // Transform function graph between
   // x1 and x2 into an object of type L2d.

// Methods for painting and calculating the area
// between function graph and x-axis.
void PaintAreaUnderGraph( Color col, Real x1, Real x2, int n,
   Boolean check_zeros = false );
Real CalcAreaUnderGraph( Real x1, Real x2, int n );
StrL2d Tangent( Real x ); // Return tangent to function graph.
```

Sample code for better understanding:

```
// Compare file "function2d.cpp"!

#include "opengeom.h"
#include "defaults2d.h"
#include "function2d.h"

Real Y( Real x )
{
   return sin ( x - PI / 4 ) + cos ( 2 * x ) + 0.1 * x;
}
Function F;

void Scene::Init( )
{
   F.Def( Y );
}
void Scene::Draw( )
{
   const Real xmin = -8, xmax = 8;
   F.CalcZeros( xmin, xmax, 100 );

   P2d P( F.Solution( 1 ), 0 );
   P2d Q = P + 2.5 * F.VelocityVector( P.x );
   DrawArrow2d( Blue, P, Q );
   Q = P + 2.5 * F.AccelerationVector( P.x );
   DrawArrow2d( Green, P, Q );

   Real x1 = F.Solution( 5 ), x2 = F.Solution( 6 );
   Real area = F.CalcAreaUnderGraph( x1, x2, 100 );
   PrintString( Black, 2, -2.3, "shaded area...%2.6f", area );
   F.PaintAreaUnderGraph( Gray, x1, x2, 100 );

   ShowAxes2d( Black, -9, 10, -3, 5 );
   F.Draw( Black, xmin, xmax, 100, MEDIUM );
   F.MarkZeros( Red, 0.1, 0.05 );
}
```

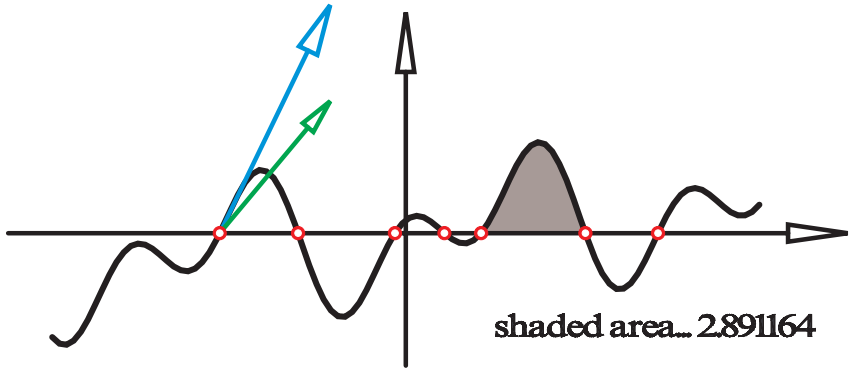

FIGURE 6.6. A 2D function graph.

class *KochFractal*; → declaration in `"fractal.h"`

An *abstract class* for the generation of fractals of Koch type. The user has to derive his own class from this class and must implement the virtual function Generator( ). Each call to Generator( ) will develop the fractal by one step. Base class.

Constructors:

KochFractal( ) { } // The only default constructor.

Definition:

void Def(**int** initial_size, *P2d* Q []);
// initial_size is the size of the starting teragon Q[ ].

Frequently used methods and operators:

```
void SetOrder( int order );
   // Redefine everything.
P2d GetPoint( int i );
   // Return point of current teragon.
void SetPoint( int i, P2d Q );
   // Redefine point of teragon.
void Draw( Color color, ThinOrThick thickness );
void MarkPoints( Color color, Real radius1, Real radius0 );
   // Mark points of teragon.
int GetStep( void );
   // Return current step of development.
void IncreaseStep( void );
virtual void Generator( void ) = NULL;
   // Virtual function that is iterated in order to develop
   // the fractal. It is applied to one side of the teragon
   // and inserts a couple of new teragon points. Must be
   // implemented by the user!
```

Sample code for better understanding:

```
class MyFractal: public KochFractal
{
public:
   virtual void Generator( void )
   {
      int n = 3; // Number of new points to each segment.

      // Do not change the following!
      int n1 = n + 1;
      int size1 = Size( ) - 1;
      int new_size = size1 * n + Size( );
      int i, j;

      P2d *Q = new P2d [Size( )];
      for ( i = 0; i < Size( ); i++ )
         Q[i] = GetPoint( i );
      O2d::Def( NoColor, new_size );

      for ( i = 0, j = 0; i < size1; i++, j += n1 )
      {
         pPoints[j] = Q[i];
         // Here you can write your own function assigning
         // n points pPoints[j+1]...pPoints[j+n] to the
         // segement Q[i]Q[i+1].
         V2d v( Q[i], Q[i+1] );
         v /= 3;
         pPoints[j+1] = pPoints[j] + v;
         pPoints[j+3] = pPoints[j+1] + v;
         v.Rotate( 60 );
         pPoints[j+2] = pPoints[j+1] + v;
      }

      pPoints[new_size-1] = Q[size1];
      IncreaseStep( );
      delete [] Q;
   }
};
MyFractal KochCurve;

P2d P[2];
P[0]( -10, -3 );
P[1]( 10, -3 );
KochCurve.Def( 2, P );

Silent( ); // To avoid the message
           // "warning: more than 30000 points"
```

```
// Develop fractal (five steps):
int i;
for ( i = 0; i < 5; i++ )
   KochCurve.Generator( );
KochCurve.Draw( Red, THIN );
```

class *L2d*; → declaration in `"lines2d.h"`

Describes a line (arbitrary polygon) in 2D. Derived from *O2d*. Base class for other frequently used classes (*ParamCurve2d*, *ClassCurve2d*, *PathCurve2d*).

Constructors:

```
L2d( ) // Default constructor.
L2d( Color col, int n ) // Specify color and number of points.
```

Methods::

```
void Draw( ThinOrThick thick, Real max_dist = -1 );
   // For max_dist > 0 a line segment is drawn only,
   // when the two adjacent points have a smaller distance.
   // This is useful for the drawing of curves with points at
   // infinity (like hyperbolas).
void DrawPartInsideCircle( const P2d &M, Real rad,
                           ThinOrThick style );
   // Draws only part inside circle with center M and radius rad.
V2d Tangent( int i ); // Approximates the tangent in the i-th point.
void Close( ) // The curve will be drawn closed.
```

class *NUBS2d*; → declaration in `"nurbs.h"`

Describes an integral B-spline curve in 2D by a control polygon and a knot vector. The curve points are computed by means of the Cox–De Boor algorithm. We provide methods for defining and displaying the curve. Dynamic memory allocation is handled internally. The base class is *ParamCurve2d*. Have a look at page 150 in Chapter 2 for more detailed information.

Constructors:

```
NUBS2d( ) { T = A = NULL; D = E = NULL; }
// Set all pointers to NULL. T is the knot vector, D the control
// polygon. A and E are used for internal computations.
```

Definition:

```
void Def( Color c, int size, int knum, Real K[], int pnum, P2d P[]);
void Def( Color c, int size, int pnum, P2d P[], int continuity,
   Boolean closed = false );
```

The first defining method allows the user to set the knot vector as well as the control points according to her/his wishes. The second method set the knot vectors automatically in order to ensure C^k-continuity (k = continuity). Furthermore, the user can choose between open and closed B-splines.

Frequently used methods and operators:

```
virtual P2d CurvePoint( Real u ) { return DeBoor( u ); }
// Overwrite the CurvePoint(...) function of ParamCurve2d.
void MarkControlPoints( Color c, Real rad1, Real rad0 = 0 );
void DrawControlPolygon( Color c, ThinOrThick style );
Real GetKnot( int i ) {return T[i]; };
P2d GetControlPoint( int i ) { return D[i]; };
int GetKnotNum( ) { return M; }
int GetPointNum( ) { return N; }
```

Sample code for better understanding:

```
NUBS2d Spline;
const int number_of_control_points = 5;

P2d Point[number_of_control_points];
Point[0].Def( -10, 0 );
Point[1].Def( -6, 6 );
Point[2].Def( -2, 0 );
Point[3].Def( -2, -6 );
Point[4].Def( -5, -6 );

const int total_size = 49;
const int continuity_class = 2;

Spline.Def( Black, total_size, number_of_control_points,
    Point, continuity_class );
Spline1.Draw( THICK );
Spline1.DrawControlPolygon( Blue, THIN );
Spline1.MarkControlPoints( Blue, 0.2, 0.1 )
```

class *NURBS2d*; → declaration in `"nurbs.h"`

Describes a rational B-spline curve in 2D by a control polygon, an array of weights, and a knot vector. The curve points are computed by applying the Cox–De Boor algorithm in 3D and projecting the result into a plane. We provide methods for defining and displaying the curve. Dynamic memory allocation is handled internally. The base class is *ParamCurve2d*. Have a look at page 150 in Chapter 2 for more detailed information.

Constructors:

```
NURBS2d( ) { T = A = W = X = NULL; D = E = NULL; }
// Set all pointers to NULL. T is the knot vector, D the control
// polygon, W the weight vector. A, X, and E are used for internal
// computations.
```

Definition:

```
void Def( Color c, int size, int knum, Real K[], int pnum, P2d P[],
    Real weight[] );
void Def( Color c, int size, int pnum, P2d P[], Real weight[],
    int continuity, Boolean closed = false );
```

The first defining method allows the user to set the knot vector as well as the control points and weights according to her/his wishes. The second method sets the knot vectors automatically in order to ensure C^k-continuity (k = continuity). Furthermore, the user can choose between open and closed B-splines.

Frequently used methods and operators:

```
virtual P2d CurvePoint( Real u ) { return DeBoor( u ); }
// Overwrite the CurvePoint(...) function of ParamCurve2d.
void MarkControlPoints( Color c, Real rad1, Real rad0 = 0 );
void DrawControlPolygon( Color c, ThinOrThick style );
Real GetKnot( int i ) {return T[i]; };
P2d GetControlPoint( int i ) { return D[i]; };
Real GetWeight( int i ) { return W[i]; };
int GetKnotNum( ) { return M; }
int GetPointNum( ) { return N; }
```

Sample code for better understanding:

```
NURBS2d Spline;
const int number_of_control_points = 5;

P2d Point[number_of_control_points];
Point[0].Def( -10, 0 );
Point[1].Def( -6, 6 );
Point[2].Def( -2, 0 );
Point[3].Def( -2, -6 );
Point[4].Def( -5, -6 );

Real weight[number_of_control_points];
int i;
for ( i = 0; i < number_of_control_points; i++ )
    weight[i] = 1;
weight[3] = 5;

const int total_size = 49;
const int continuity_class = 2;

// create a closed spline curve
Spline.Def( Black, total_size, number_of_control_points,
    Point, weight, continuity_class, true );

Spline.Draw( THICK, 5 );
Spline.DrawControlPolygon( Blue, THIN );
Spline.MarkControlPoints( Blue, 0.2, 0.1 );
```

class *O2d*; → declaration in `"o2d.h"`

Describes a conglomerate of points in 2-space (the points, however, are stored as 3D points with $z = 0$). Derived from *O23d*.

Constructors:

O2d( ) // Default constructor,

Definition:

void Def(*Color* f, **int** n)
void Def(*Color* f, **int** n, *Coord2dArray* P)
void SameAs(*O2d* *obj);

Geometric transformations (apply to all objects derived from O2d!):

void Translate(*Real* dx, *Real* dy);
void Translate(**const** *V2d* &v);
void Scale(*Real* kx, *Real* ky = 0);

```
void Rotate( const P23d &center, Real w );
void Rotate( );
  // Applies a predefined standard rotation to the object.
  // Faster than computing the same rotation matrix for every
  // single point (compare Example 4).
void Reflect( const StrL2d &strLine );
```

Manipulating the element points:

```
void SetPoint( int i, Real x0, Real y0 );
void AssignVertices( Coord2dArray koord );
```

class *P2d*; → declaration in `"points.h"`

Describes a 2D point (x, y). Inherits the methods and operators of *P23d* and thus also those of *V23d*, *V2d* and *V3d*.

Constructors:

```
// See constructors of the class V2d!
```

Additional operators:

```
inline friend P2d operator + ( const P2d &v, const V2d &w )
inline friend P2d operator - ( const P2d &v, const V2d &w )
  // Allows to add or subtract vectors to a point.
inline friend P2d operator * ( const Real t, const P2d &v )
  // Scaling of a point. Overwrites vector scaling
inline friend V2d operator + ( const P2d &v, const P2d &w )
inline friend V2d operator - ( const P2d &v, const P2d &w )
  // Add or subtract points like vectors.
P2d & operator = ( const P2d &P )
P2d & operator = ( const V2d &v )
P2d & operator = ( const P23d &v )
  // Make these types compatible.
```

Additional methods:

```
void Def( Real x0, Real y0 );
  // Set coordinates to (x0, y0)
void Rotate( );
  // Apply a predefined 3D rotation in order to speed up
  // the code (compare Example 4).
void Rotate( const P2d& center, Real angle_in_deg );
  // Rotate point about a center through an angle given in degrees.
void Translate( Real dx, Real dy );
  // Translate point by means of the vector (dx, dy).
void Translate( const V2d &v );
  // Translate point by means of the vector v.
void Scale( Real kx, Real ky );
  // Scale point in two directions (affine transformation).
void Write( Color col, Real x0, Real y0, char * text, int size );
  // This will write the coordinates of the point in color col
```

```
    // at the position (x0, y0) in a legible manner:
    // 'text(x, y)' (3 digits after the comma).
    // Default character size is 1 unit.
  void AttachString( Color col, Real dx, Real dy, char * text );
    // Write output in color col at the position (x + dx, y + dy).
    // You can write almost everything by means of C-syntax.
  P2d Center( P23d &P );
    // Return the center (midpoint) between the
    // point and the point P.
  Real PolarAngleInDeg( ) const;
    // See V2d.
  void Reflect( const StrL2d &g );
    // Reflect the point at a straight line g.
  void MarkPixel( Color f ) const;
    // Mark the pixel at the position of the point.
  Boolean Collinear( const P2d &P, const P2d &Q,
    Real tol = 1e-8 ) const;
    // Check whether three points are collinear.
    // tol is the accuracy limit.
  Real Dist( const StrL2d &s ) const;
    // Return the oriented distance to the straight line s.
  inline P2d Mid( const P2d P );
    // Inline version of Center( ) method.
```

class *ParabolaOfNthOrder*; → declaration in `"parabola_n.h"`

For some mathematical calculations, parabolas of n-th order may be required. It is given by the function of the graph

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0 = \sum_{k=0}^{n} a_k x^k$$

The quadratic parabola was implemented already in version 1. The class is derived from *ParamCurve2d*. Apart from the constructor and the destructor, the class has the following methods that are of interest for the user:

Definition:

```
void Def( Color c, int n_points, Real umin, Real umax,
  int n_controls, const Coord2dArray p );
void Def( Color c, int size, Real umin, Real umax,
      int n_controls, const P2d p[] );
```

A parabola of n-th order is given by $n+1$ "control points" (either given by a *Coord2dArray* or an array of points *P2d*. Such a curve is a function graph; the leftmost point stems from the parameter umin, the rightmost from umax. The parameter size fixes the number of desired points. Since the class is derived from *ParamCurve2d*, it inherits all public methods

of this class. Additionally, the following member functions are implemented:

Frequently used methods and operators:

```
P2d GetP( int i ); // Return the i-th control point.
Real GetCoeff( int i );
   // Return the i-th coefficient of the equation.
void GetCoeff( int &i, Real c[] );
   // Return the i-th coefficient of the equation and the
   // corresponding array of coefficients
int GetOrder( ); // Return the order.
void MarkGivenPoints( Color c, Real r1, Real r2 );
   // Mark control points.
void Delete( ); // Free dynamically allocated memory
   // (automatically called by the destructor).
```

Figure 6.7 shows the difference between a cubic spline and a parabola of n-th order (the figure was created by means of the program `"parab_nth_order.cpp"`. The code is so short that we give a listing:

Sample code for better understanding:

```
ParabolaOfNthOrder Par;
CubicSpline2d Spline;
ShowAxes2d( DarkGray, -3.5, 12, -4, 4 );
Coord2dArray P =
   { {-3, 1}, {-1.5, 0}, {0, 1}, {2, 1}, {4, 3}, {7, -1}, {9, -1} };
Par.Def( Blue, 100, -3.3, 9.3, 7, P );
Par.Draw( MEDIUM );
Spline.Def( Red, 7, P, 8 );
Spline.Draw( THICK );
Par.MarkGivenPoints( Red, 0.17, 0.1 );
```

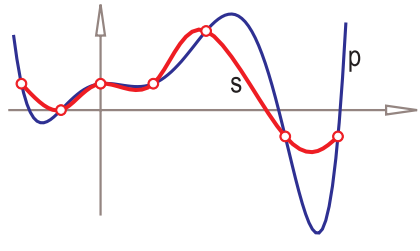

FIGURE 6.7. A cubic spline s and a parabola p of n-th order.

class *ParamCurve2d*; → declaration in `"lines2d.h"`

An *abstract class* for a smooth 2D curve, defined by a parametric representation. Derived from *O2d* and *L2d*. In order to define a specific curve, the user has to derive an additional class from this class!

Definition:

```
void Def ( Color f, int n, Real umin, Real umax );
// umin and umax are the parameter limits for the curve.
// The program will calculate n points on the curve, corresponding
// to evenly distributed parameter values in between.
```

Additional member variables and methods:

```
Real u1, u2; // Parameter limits.
virtual P2d CurvePoint( Real u ) = 0; // dummy
virtual V2d TangentVector( Real u );
// Return the approximated tangent vector at parameter u.
virtual V2d NormalVector( Real u );
// Return the approximated normal vector at parameter u.
virtual StrL2d Tangent( Real u );
// Return the approximated tangent at parameter u.
virtual StrL2d Normal( Real u );
// Return the approximated normal at parameter u.
virtual Real RadiusOfCurvature( Real u );
// Return the radius of curvature at parameter u.
virtual void GetOsculatingCircle( Real u, Color col, Circ2d &osc_circ );
// Define the osculating circle at parameter u.
virtual Real ArcLength( Real u1, Real u2, int accuracy = 300 );
// Return the arc length ∫_{u1}^{u2} sqrt(ẋ² + ẏ²) du.
// By default, the curve is interpolated by a 300-sided polygon.

// Methods to compute curves that are associated with the
// parametric curve. The result is stored in the L2d
// argument (compare Section 2.1, page 40).
virtual void GetEvolute( Real u1, Real u2, L2d &evolute );
virtual void GetCata( const P2d &pole, Real u1, Real u2,
                      L2d &evolute );
virtual void GetPedal( const P2d &pole, Real u1, Real u2,
                       L2d &pedal );
virtual void GetAntiPedal( const P2d &pole, Real u1, Real u2,
                           L2d &antipedal );
virtual void GetOrtho( const P2d &pole, Real u1, Real u2,
                       L2d &orthonomic );
virtual void GetAntiOrtho( const P2d &pole, Real u1, Real u2,
                           L2d &orthonomic );
virtual void GetOffset( Real distance, Real u1, Real u2,
                        L2d &offset );
virtual void GetInvolute( Real param_value, Real u1, Real u2,
                          L2d &involute );
```

Sample code for better understanding:

```
// Compare file "USER/TEMPLETS/paramcurve2d.cpp"!

class MyCurve: public ParamCurve2d
{
public:
    P2d CurvePoint( Real u )
    {
        // a circle of radius 5:
        const Real r = 5;
        return P2d( r * cos( u ), r * sin( u ) );
    }
};
MyCurve Curve;
void Scene::Init( )
{
    int number_of_points = 101;
    Real param1 = -PI, param2 = PI;
    Curve.Def( Black, number_of_points, param1, param2 );
}
void Scene::Draw( )
{
    Curve.Draw( THICK );
}
```

class *PathCurve2d*; → declaration in `"lines2d.h"`

A curve of unknown size. Derived from *O2d* and *L2d*. Used for the determination of path curves during kinematic motions or animations.

Constructors and Definition:

```
PathCurve2d( Color c = Gray, int max_points = MAX_POLY_SIZE,
             Real critical_distance = 0 );
void Def( Color c = Gray, int max_points = MAX_POLY_SIZE,
          Real critical_distance = 0 );
  // Allows max_points on the path curve. Path will be
  // considerd to be 'closed' when the first and last points have a
  // distance less than critical_distance.
```

Additional methods:

```
void AddPoint( const P2d &P );
  // Add point P to the curve.
```

class *Poly2d*; → declaration in `"poly2d.h"`

Closed, convex polygon in 2-space. Derived from *O2d*. Parent class for many other 2D classes (*RegPoly2d*, *Rect2d*...). For non convex polygons use the class *ComplexPoly2d*.

Constructors:

Poly2d( )
Poly2d(*Color* col, **int** numOfPoints, *FilledOrNot* filled = *EMPTY*)
Poly2d(*Color* col, **int** numOfPoints, *Coord2dArray* points,
FilledOrNot filled = *EMPTY*)

Definition:

void Def(*Color* col, **int** numOfPoints, *FilledOrNot* filled = *EMPTY*);
void Def(*Color* col, **int** numOfPoints, *Coord2dArray* points,
FilledOrNot filled = *EMPTY*);

Methods and operators:

void Draw(*ThinOrThick* thick);
// Draw the outline.
void Outline(*Color* c, *ThinOrThick* style);
// Draw the outline in the specified color.
void Shade( );
// Plot the filled polygon.
void ShadeWithContour(*Color* col, *ThinOrThick* thick,
Real alpha_value = 1);
// A combination of Draw and Shade. Additionally, the color col
// of the outline can be specified.
void ShadeWithTexture(*TextureMap* &Map,
Real ScaleX = 1, *Real* ScaleY = 1,
Real RotAngleInDeg = 0,
Real TranslateX = 0, *Real* TranslateY = 0,
Boolean repeat = true);
// Method to map a texture on the polygon (compare
// Section 4.4).

class *Projectivity2d*; → declaration in `"proj_geom.h"`

This class consists of two classes of type *ProjScale2d* and provides methods to deal with the projectivity between them. Note that some of the methods make sense only if the projective scales are of the right type (*PencilOfLines2d*, *PencilOfPoints2d*, *PointsOnConic2d*, *TangentsOfConic2d*). Base class. For further information, please consult Section 5.2.

Constructors:

Projectivity2d( ); // Default constructor.

Definition:

```
void Def( ProjScale2d &InScale1, ProjScale2d &InScale2,
        const Real eps = 1e-6, const Real infinity = 1e6 );
   // Cross ratios of absolute value eps or less will
   // be treated as zero. Cross ratios of absolute value
   // infinity will be set to infinity.
```

Methods and operators:

```
P2d MapPointToPoint( const P2d &A );
StrL2d MapPointToLine( const P2d &A );
P2d MapLineToPoint( const StrL2d &a );
StrL2d MapLineToLine( const StrL2d &a );
   // Four methods to map elements of the first scale
   // to elements of the second scale. The method to be
   // used depends on the type of the projective scales.
P2d InvMapPointToPoint( const P2d &A );
StrL2d InvMapPointToLine( const P2d &A );
P2d InvMapLineToPoint( const StrL2d &a );
StrL2d InvMapLineToLine( const StrL2d &a );
   // Four methods to map elements of the second scale
   // to elements of the first scale. The method to be
   // used depends on the type of the projective scales.
void MarkElements( Color col, const Real rad1,
     const Real rad2 = 0 );
void MarkElements( Color col[3], const Real rad1,
     const Real rad2 = 0 );
   // Two methods to mark the base points on either of the
   // two projective scales, either in one color or in three
   // different colors according to the respective kind of point
   // (origin, unit "point," "point" at infinity).
void ConnectElements( Color col, ThinOrThick thick );
void ConnectElements( Color col[3], ThinOrThick thick );
   // Two methods to connect corresponding base points on the
   // two projective scales. The connecting lines will be either
   // drawn in one color or in three different colors according
   // to the respective kind of point (origin, unit "point",
   // "point" at infinity).
P2d IntersectLines( Real ratio );
   // Method to intersect corresponding lines.
StrL2d ConnectingLine( Real ratio );
   // Method to connect corresponding points.
ProjType2d GetType1( ) { return Type1; }
ProjType2d GetType2( ) { return Type2; }
   // Two methods to return the respective types of the projective
   // scales. Possible return values are PencilOfLines2d,
   // PencilOfPoints2d, PointsOnConic2d and TangentsOfConic2d.
```

Sample code for better understanding:

```
ProjScale2d Scale1, Scale2;
Projectivity2d Proj;
Conic Conic1;
Color Col[3] = { Red, Green, Blue };
P2d P1[3];
P1[0].Def( -5, 6 ), P1[1].Def( 5, 6 ), P1[2].Def( 0, 12 );
Conic1.Def( Black, 150, P1, Origin, Xaxis2d );
P2d P2[5];
P2[0].Def( 15, 0 ), P2[1].Def( -15, 0 ), P2[2].Def( 0, 0 );
Scale1.Def( Conic1, P1 );
Scale2.Def( Xaxis2d, P2 );
Proj.Def( Scale1, Scale2 );
```

class *ProjScale2d*; → declaration in `"proj_geom.h"`

Describes a projective scale on either a pencil of points, a pencil of lines, the point set on a conic, or the tangent set of a conic. The scale consists of a supporting element (straight line, point, or conic) and a projective coordinate system (origin, "point" at infinity and unit "point").[1] Base class. For further information, please consult Section 5.2.

Constructors:

```
ProjScale2d( ); // Default constructor.
```

Definition:

```
ProjScale2d & operator = ( ProjScale2d &other );
void Def( const P2d &V, StrL2d t[3] ); // pencil of lines
void Def( const StrL2d &l, P2d Q[3] ); // pencil of points
void Def( Conic &c, P2d Q[3] ); // points on conic
void Def( Conic &c, StrL2d t[3] ); // tangents of conic
```

Methods and operators:

```
void Draw( Color col, Real u0, Real u1, ThinOrThick thick );
   // Drawing method for pencil of points.
void Draw( ThinOrThick thick );
   // Drawing method for points or tangents of conic.
void Mark( Color col, Real rad1, Real rad2 = 0 );
   // Drawing method for pencil of lines.
void MarkElement( Color col, Real rad1, Real rad2, int i );
void MarkElements( Color col, Real rad1, Real rad2 );
```

[1] In this context the word "point" means either an actual point or a straight line (principle of duality).

```
    // Two methods for marking origin (i = 0), unit point (i = 1),
    // and point at infinity (i = 2) if these elements are points.
void DrawElement( Color col, Real u0, Real u1,
        ThinOrThick thick, int i );
void DrawElements( Color col, Real u0, Real u1,
        ThinOrThick thick );
    // Two methods for drawing origin (i = 0), unit point (i = 1),
    // and point at infinity (i = 2) if these elements are straight lines.
ProjType2d GetType( ) { return Type; }
    // Return the type of the projective scale. Possible return values
    // are PencilOfLines2d, PencilOfPoints2d, PointsOnConic2d,
    // and TangentsOfConic2d.
void SetElement( const StrL2d &t, int i )
        { s[i] = t; ComputeReals( ); }
void SetElement( const P2d &Q, int i )
        { P[i] = Q; ComputeReals( ); }
    // Two methods for redefining the projective scale by setting
    // the origin, unit "point" or "point" at infinity.
    // ComputeReals( ) is a private method of the class that
    // computes some necessary data.
P2d GetPoint( int i ) { return P[i]; }
StrL2d GetLine( int i ) { return s[i]; }
    // Two methods for returning origin, unit "point"
    // or "point" at infinity.
P2d PointOfCrossRatio( const Real ratio );
StrL2d LineOfCrossRatio( const Real ratio );
    // Return the point or straight line, respectively, that
    // determines the given cross ratio with respect to the
    // projective scale.
Real CrossRatio( const P2d &A, const Real eps = 1e-6,
        const Real infinity = 1e6 );
Real CrossRatio( const StrL2d &a, const Real eps = 1e-6,
        const Real infinity = 1e6 );
    // Return the cross ratio that a given straight line or point
    // determines on the projective scale. If its
    // absolute value is smaller than eps or larger than infinity,
    // the return values will be 0 or infinity itself.
```

Sample code for better understanding:

```
ProjScale2d Scale
StrL2d s( P2d( 0, −10 ), Xdir2d );
P2d S[3];
S[0] = s.InBetweenPoint( 0 );
S[1] = s.InBetweenPoint( −5 );
S[2] = s.InBetweenPoint( 10 );
Scale.Def( s, S );
Scale.Draw( Green, −15, 15, VERY_THICK );
Scale.MarkElements( Green, 0.2, 0.1 );
P2d Point;
Real ratio = −1;
Point = Scale.PointOfCrossRatio( ratio );
Point.Mark( Red, 0.2, 0.1 );
if ( !( Scale.CrossRatio( Point ) == −1 ) )
    Write( "numerical problem" );
```

class *RatBezierCurve2d*; → declaration in `"bezier.h"`

Describes a rational Bézier curve in 2D by its control polygon and its weights. The curve points are computed by means of DeCasteljau's algorithm. We provide methods for manipulating the curve (changing of control points and weights, degree elevation, splitting). The base class is *ParamCurve2d*. Some methods are overwritten. Have a look at page 126 in Chapter 2 for more detailed information.

Constructors:

```
RatBezierCurve2d( ) { B = NULL; C = NULL; w = NULL; }
// B and C are dynamically allocated arrays of control points,
// w is a dynamically allocated array of reals.
```

Definition:

```
void Def( Color c, int total_size, int size_of_control_poly,
          P2d Control[], Real weigth[] );
void Def( Color c, int total_size, int size_of_control_poly,
          Coord2dArray Control, Real weight[] );
// total_size is the number of curve points that will
// be computed while size_of_control_poly is the number
// of control points to be used.
```

Frequently used methods and operators:

```
virtual P2d CurvePoint( Real u );
   // Compute curve point by means of DeCasteljau's algorithm.
virtual StrL2d Tangent( Real u )/* const = 0 */;
void DrawControlPolygon( Color col, ThinOrThick thickness );
```

```
void MarkControlPoints( Color col, Real rad1, Real rad0 );
void SetControlPoint( int i, P2d P );
   // Redefine control point with index i.
void SetWeight( int i, Real w );
   // Redefine weight with index i.
P2d GetControlPoint( int i );
   // Return control point with index i.
Real GetWeight( int i );
   // Return weight with index i.
int GetPNum( ) { return PNum; }
   // PNum is the number of control points
void ElevateDegree( int i ); // Elevates the degree by i.
void ReversePolygon( void );
   // Reverse the order of the control points
   // (useful after splitting of the curve).
void Split( Real u, Boolean second_half );
   // Split the Bézier curve at the parameter value u.
   // Depending on the value of second_half, the new
   // curve is identical to the part of the old curve that
   // corresponds to the parameter interval [0, u] or [u, 1].
```

Sample code for better understanding:

```
RatBezierCurve2d BezierCurve;
P2d P[3];
P[0]( -7, -3 ), P[1]( -4, 2 ), P[2]( 3, 3 );
Real weight[3];
weight[0] = 2, weight[1] = 1, weight[2] = 3;
BezierCurve.Def( Black, 200, 3, BasePoints, weight );
BezierCurve.ElevateDegree( 2 );
BezierCurve.Split( 0.5, true );
BezierCurve.Draw( VERY_THICK );
BezierCurve.DrawControlPolygon( Blue, THICK );
BezierCurve.MarkControlPoints( Blue, 0.15, 0.1 );
```

class *Rect2d*; → declaration in `"poly2d.h"`

Rectangle in 2-space. Inherits methods of *Poly2d*. In the starting position two rectangle sides lie on the x- and y-axes of the coordinate system and one vertex coincides with the origin.

Constructors and definition:

```
Rect2d( ){}
Rect2d( Color c, Real length, Real width,
    FilledOrNot filled = EMPTY )
void Def( Color c, Real length, Real width,
    FilledOrNot filled = EMPTY );
void Def( Color c, const P2d &LowerLeft, const P2d &UpperRight,
    FilledOrNot filled = EMPTY );
```

class *RegPoly2d*; → declaration in `"poly2d.h"`

Closed regular polygon in 2-space. Derived from *Poly2d*.

Definition:

```
void Def( Color col, const P2d &FirstElem, const P2d &mid,
          int n, FilledOrNot filled = EMPTY );
  // Center and a vertex are given.
void Def( Color col, const P2d &mid, Real rad,
          int n, FilledOrNot filled = EMPTY );
  // Center and radius are given.
  // First vertex lies on line parallel to the x-axis through the center.
void Def( Color col, const P2d &vertex1, const P2d &vertex2,
          int n, int orientation = 1, FilledOrNot filled = EMPTY );
  // Two adjacent vertices and the orientation (+1 or −1) are given.
```

Methods:

```
P2d GetCenter( ) const; // Return the center.
Real GetRadius( ) const; // Return the radius.
```

class *Rod2d*; → declaration in `"poly2d.h"`

Describes a rod in 2-space. A rod may be imagined as a polygon with two vertices or a line segment. It is used mainly for kinematic animations.

Definition:

```
void Def( Color col, Real x1, Real y1, Real x2, Real y2 );
void Def( Color col, const P2d &P, const P2d &Q );
```

Additional methods (see also Poly2d):

```
void Draw( ThinOrThick thick );
void Dotted( int n, Real rad );
void LineDotted( int n, ThinOrThick thick );
```

Sample code for better understanding:

```
Rod2d rod;
rod.Def( Magenta, -5, 0, 0, 5 );
rod.Draw( THICK );
rod.Rotate( Origin, 45 );
rod.Dotted( 20, 0.05 );
```

class *Sector2d*; → declaration in `"arc2d.h"`

Class for drawing a sector of a 2D circle. It provides various defining methods and "getters" for objects related to the sector. The sector is either filled or not, and the user can decide whether to draw the line segments that connect the center with start and end point (value of only_segment = **false** or **true**; compare Figure 6.8). *Sector2d* is derived from *Sector3d* and inherits a few useful methods.

Constructors:

```
Sector2d( ) { }; // Default constructor.
```

Definition:

```
void Def3Points( Color col, const P2d &P, const P2d &Q,
   const P2d &R, int size, FilledOrNot filled = EMPTY );
   // Define sector via three points. The argument list is identical to
   // that of the last Def(...) method. Therefore, the name of
   // this method differs a little from standard OPEN GEOMETRY
   // conventions. size is the number of points on the arc that will
   // be computed.
void Def( Color col, const P2d &FirstElem, const P2d &Center,
   Real central_angle_in_deg, int size, FilledOrNot = EMPTY );
   // Define sector via start point, center, and central angle. Note that
   // the apex angle will always be less than 180°!
void Def( Color col, const P2d &FirstElem, const P2d &Center,
   const P2d &LastElem, int size, FilledOrNot = EMPTY );
   // Define sector via start point, center, and end point.
```

Frequently used methods and operators:

```
void Draw( Boolean only_segment, ThinOrThick thick );
// If only_segment is false, the connecting lines of center
// with start and end point will be displayed.
void Shade( );
// Shades the sector; has the same effect as Draw if the sector has
// been defined as FILLED.
void ShadeWithContour( Color c, ThinOrThick thick,
   Real alpha_value = 1 );
// The same as Shade; additionally, the contour is displayed.
```

```
// The meaning of the remaining methods should be obvious:
P2d GetCenter( );
P2d GetFirstPoint( );
P2d GetLastPoint( );
P2d GetPointOnBisectrix( );
```

Sample code for better understanding:

```
Sector2d Arc1, Arc2, Arc3;
P2d M1, M2;
P2d A;
M1( 0, 5 );
M2( 5, -1 );
Arc1.Def( Red, P2d( -3, 0 ), M1, M2, 30, EMPTY );
A( -20, 0 );
Arc2.Def( Black, Arc1.GetLastPoint( ), M2, A, 30 );
Arc3.Def3Points( Green, Arc1.GetFirstPoint( ), Arc1.GetLastPoint( ),
    Arc2.GetLastPoint( ), 30 );
Arc1.Draw( MEDIUM );
Arc2.Draw( MEDIUM );
Arc3.Draw( THIN );
Arc1.GetFirstPoint( ).Mark( Green, 0.2 );
Arc1.GetLastPoint( ).Mark( Orange, 0.2 );
Arc2.GetLastPoint( ).Mark( Blue, 0.2 );
StraightLine2d( Blue, M1, M2, THIN );
M1.Mark( Red, 0.2, 0.12 );
M2.Mark( Black, 0.2, 0.12 );
```

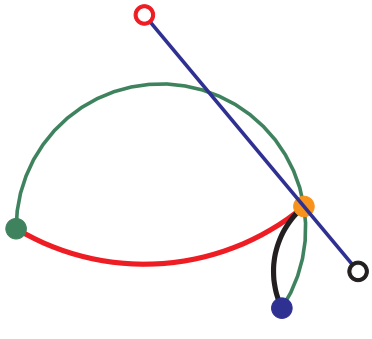

FIGURE 6.8. Different 2D sectors.

class *SliderCrank*; → declaration in `"kinemat.h"`

Class to describe a slider crank. Derived from *FourBarLinkage*. The class is equipped with methods for computing and displaying special

positions (compare Figure 6.9 and [14], Chapter 9). Requires the inclusion of `"kinemat.h"` at the beginning of the program.

Definition:

```
void Def( Real LA, Real AB, Real Lb, Real start_alpha_in_deg,
          Real delta_alpha, Boolean take_left = false );
  // B runs on straight line b (distance Lb from L)
  // delta_alpha is the step in the animation.
void Def( const P2d &L, const P2d &A, const P2d &B );
```

Methods:

```
void Def( Real LA, Real AB, Real Lb, Real start_alpha_in_deg,
          Real delta_alpha, Boolean take_left = false );
  // B runs on straight line b (distance Lb from L)
  // delta_alpha is the step in the animation.
void Def( const P2d &L, const P2d &A, const P2d &B );
```

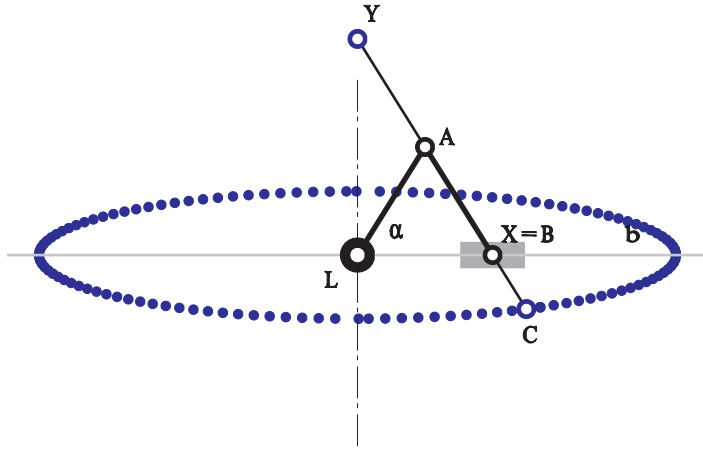

FIGURE 6.9. A slider crank.

class *StrL2d*; → declaration in `"strl2d.h"`

StrL2d describes an oriented straight line in 2-space. It has already been explained in [14], p. 80. However, there are quite a few new methods.

Constructors:

```
StrL2d( );  // Dummy constructor. No initialization
StrL2d( const P23d &P, const V23d &r );
  // The line is given by a point and a direction vector.
  // The direction vector will be normalized in any case.
  // See also method Def( ).
StrL2d(  const P2d &P, const P2d &Q );
```

```
    // The line is given by two points.
```

Definition:

```
void Def( const P23d &P, const V23d &r );
    // Define line by means of a point and a direction vector:
    // x⃗ = P⃗ + λr⃗
    // The vector need not be normalized (it will be normalized to a
    // direction vector r⃗0).
    // Internally, the line has the equation x⃗ = P⃗ + t r⃗0.
    // 3D points and 3D vectors are interpreted two-dimensionally.
void Def( const P2d &P, const P2d &Q );
    // Define line by means of two points. Internally, the line has
    // the equation x⃗ = P⃗ + t r⃗0 with r⃗ = Q⃗ − P⃗.
```

Operators:

```
friend P2d operator * ( const StrL2d &g1, const StrL2d &g2 );
    // Intersection point of two straight lines g1 and g2.
    // When the lines are parallel, a "point of infinity" is taken,
    // when the lines are identical, the result is an arbitrary point
    // on the line. See also method SectionWithStraightLine( )
Boolean operator == ( const StrL2d &other ) const;
    // Are two lines identical from the mathematical point of view?
StrL2d & operator = ( const StrL2d &other );
    // Applying this operator yields two identical straight lines!
```

Frequently used methods:

```
V2d GetDir( ) const;
    // Return the normalized direction vector r⃗0.
void SetDir( const V2d &newdir );
    // Change direction vector (newdir need not be normalized)
V2d GetNormalVector( ) const;
    // Return the normalized normal vector n⃗ ⊥ r⃗0.
P2d GetPoint( ) const;
    // Return the point that was used for the definition of the line.
Real GetConst( ) const;
    // Return the constant c of the straight line nx x + ny y = c.
void SectionWithStraightLine( const StrL2d &g, P23d &P,
                              Real &t, Boolean &parallel ) const;
    // Similar to the * operator. The result, however, is more detailed:
    // The parameter t to the point (with respect to the current
    // line) is passed by reference, and the flag parallel is set.
Real OrientedDistance( P23d &P );
    // Return the oriented distance of a point. When the origin of the
    // coordinate system is not on the same side of the straight line,
    // the distance will be negative.
P2d InBetweenPoint( Real t ) const;
    // Return the point x⃗ = P⃗ + t r⃗0.
StrL2d GetNormal( const P2d &P ) const;
    // Return a straight line perpendicular to the current one,
```

```
    // coinciding with P.
StrL2d GetParallel( Real dist ) const;
    // Return a straight line parallel to the current
    // one with distance dist.
P2d NormalProjectionOfPoint( const P2d &P ) const;
    // Return the normal projection of the point P onto the line.
Boolean ContainsPoint( const P2d &P, Real tol = 1e-4 ) const;
    // Check whether a point coincides with the line, i.e., whether its distance is
    // smaller than a given tolerance.
Boolean IsParallel( const StrL2d &other, Real tol = 1e-8 ) const;
    // Check whether the lines are parallel. The angle between the direction
    // vectors must not differ more than a certain tolerance.
Real GetParameter( const P2d &P ) const;
    // Return the parameter t in the vector equation x = P + t r0.

// Some methods for the application of transformations.

void Translate( const V2d &v );
    // Translate the line by means of a 2D vector.
void Rotate( const P2d &center, Real angle_in_deg );
    // Rotate the line about a center through a given angle
    // (in degrees).
void Reflect( const StrL2d &g );
    // Reflect the line at a given line g.
int Refract( const StrL2d &refracting_line, Real ior,
    StrL2d &refracted_line ) const;
    // Refract the line at a given line refracting_line. ior is the
    // index of refraction. In contrast to the Reflect( ) method,
    // the result is stored in refracted_line. The return value tells
    // what has really happened: It is 2 if real refraction occurs,
    // 1 for total reflection, and 0 if the incoming straight line has
    // a wrong orientation. refracted_line will always have the correct
    // orientation according to physical models.

// Finally, some drawing methods:

void Draw( Color col, Real t1, Real t2, ThinOrThick thick ) const;
    // Draw the line in color col from the point that corresponds
    // to the parameter t1 to the point that corresponds to the
    // parameter t2. You have to specify the width of the line:
    // THIN, MEDIUM, THICK, VERY_THICK.
void LineDotted( Color col, Real t1, Real t2, int n,
                 ThinOrThick thick ) const;
    // Similar to the ordinary drawing.
    // Draw the line as dotted, with n intervals.
void Dotted( Color col, Real t1, Real t2, int n, Real rad ) const;
    // Similar to the ordinary drawing; draws the line dotted
    // (n 'dots', i.e., small circles with radius rad).
void Dashed( Color col, Real t1, Real t2, int n,
    Real f, ThinOrThick thick ) const;
```

```
// Similar to Dotted drawing mode; draws n short dashes.
// The dash length depends on the real f ∈ [1.01, 3]. The bigger
// f is, the shorter the dashes are.
```

class *Trochoid*; → declaration in `"kinemat.h"`

Describes the path curve of a trochoid motion (circle rolling, compare [14], p. 252 and Figure 6.10). Alternatively, the path curve is generated by two rods that rotate with constant angular velocity. The class is derived from *L2d* and requires the inclusion of `"kinemat.h"` at the beginning of the program.

Definition:

```
void Def( Color c, Real FR, Real RC,
          Real deg1, Real deg2, Real delta_deg );
  // deg1 and deg2 are the inclinations of FR and RC to
  // the x-axis (Figure 6.10); delta_deg is the angle for
  // the animation.
void Def( Color c, Real rad1, Real rad2, Real RC,
          Real deg1, Real deg2, Real delta_deg );
  // Alternative: The radii of the Cardan circles are given
```

Additional methods:

```
void ShowPoleCurves( Real deg, ThinOrThick linestyle );
  // Cardan circles.
void ShowBars( Real deg, ThinOrThick linestyle, Real r = 0,
  Boolean attach_names = true );
P2d GetF( );
P2d GetR( Real deg );
P2d GetC( Real deg );
P2d GetPole( Real deg );
P2d CenterOfOsculatingCircle( const P2d &C, Real deg );
```

class *V2d*; → declaration in `"vector.h"`

Describes a two-dimensional vector (x, y). Inherits the methods and operators of *V23d* (partly overwritten!).

Constructors:

```
V2d( ); // No initialization
V2d( Real x0, Real y0 ); // Initialization with (x0, y0)
V2d( const P2d &P, const P2d &Q ); // Init. with PQ⃗ = q⃗ − p⃗
V2d operator −( ) const; // Scaling with −1.
```

Additional or overwriting operators:

```
friend V2d operator + ( const V2d &v, const V2d &w );
  // Vector addition v⃗ + w⃗.
friend V2d operator − ( const V2d &v, const V2d &w );
```

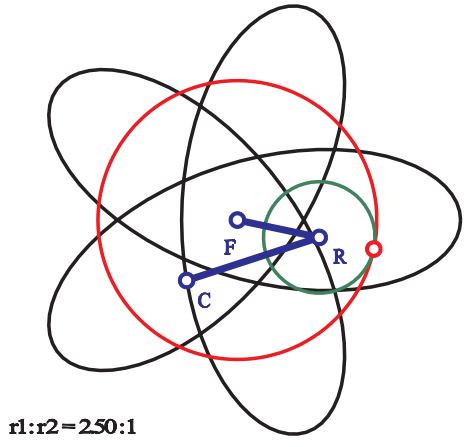

FIGURE 6.10. A trochoid.

```
    // Vector subtraction v⃗ − w⃗.
  void operator += ( const V2d &v );
    // Analog to += operator of ordinary numbers.
  void operator −= ( const V2d &v );
    // Analog to −= operator of ordinary numbers.
  friend Real operator ∗ ( const V2d &v, const V2d &w );
    // Dot product v⃗ w⃗.
  friend V2d operator ∗ ( const Real t, const V2d &v );
    // Scale vector with real number: t v⃗.
```

Additional or overwriting methods:

```
  Real PolarAngleInDeg( ) const;
    // Return the polar angle φ of the vector (−180° < φ ≤ 180°).
  V2d NormalVector( ) const;
    // Rotate the vector through +90°
  Boolean Collinear( const V2d &v, Real tol = 0 ) const;
    // Return true if the vector is collinear with v⃗, otherwise false.
    // If no tolerance is set, a tiny default tolerance is used.
  void Reflect( const StrL2d &g );
    // Reflect the vector at a straight line g.
  void Print( char *str = "@" ) const;
    // Display the components as a message: 'text = ( x, y )'.
  void Rotate( Real angle_in_deg );
    // Rotate the vector through angle (in degrees).
  Real Angle( const V2d &v,
        Boolean already_normalized = false,
        Boolean check_sign = false ) const;
    // Return the angle between two vectors (in arc length!).
    // Use Deg( Angle(...)) to get the angle in degrees.
```

6.2 Useful 3D Classes and Their Methods

class *Arc3d*; → declaration in `"arc3d.h"`

Arc3d describes a segment of a 3D circle. It is a successor of *Sector3d* and has only one additional method:

void Draw(*ThinOrThick* style, *Real* offset = *STD_OFFSET*);

This method calls the drawing method of *Sector3d without* the option of drawing the line segments that connect the arc center with start and end point.

class *ArrayOfSolids*; → declaration in `"solid.h"`

Describes an array of solids. More or less only for internal use. Better work with the more general class *Cad3Data* that is described below!

Constructors:

ArrayOfSolids( ); // Default constructor.

Definition:

void Def(**int** n0);
// Number of solids.

Operators:

Solid & **operator** [ ] (**int** i);
// Return the *i*-th solid.

Methods:

int Size( ); // Returns number of solids;
// the rest is of no importance for the user.

class *Arrow3d*; → declaration in `"arrows3d.h"`

Arrow3d is a new Open Geometry class for displaying different types of arrows in three dimensions. It is derived from *Solid* and defined via an integer value type $\in \{\pm 1, \pm 2, \pm 3, \pm 4\}$. This produces a certain standard arrow that can be scaled, rotated, or translated to its desired position. There exist also special methods for doing that. The arrows of type ± 1, ± 2, ± 3, and ± 4 are congruent. They just point in different directions. The arrows of types 1 and 2 are straight; the arrows of types 3 and 4 are curved (compare Figure 6.11).

Definition:

void Def(**int** type);

Frequently used methods and operators:

```
void Shade( Color col );
void Scale( Real k );
void Scale( Real kx, Real ky, Real kz );
void Translate( Real tx, Real ty, Real tz );
void Translate( const V3d &t );
void Rotate( const StrL3d &axis, Real angle_in_deg );
void Twist( Real angle_in_deg ); // Rotate arrow about its local axis.
void ShadeAtPosition( Color col, const P3d &apex,
   const V3d &local_x, const V3d &local_z );
   // Shade arrow at position specified by &apex, local_x and local_z.
StrL3d GetLocalAxis( );
void GetLocalSystem( V3d &x, V3d &y, V3d &z );
int GetType( ) { return arrow_type; }
```

Consider also two additional methods for displaying 3D arrows:

Listing from `"arrows3d.h"`:

```
void RoundArrow( Color c,
            const StrL3d &axis,
            Real rad, Real height,
            Real angle,
            Real angle2, Real delta_z );
void DrawArrow3d( Color col,
            const P3d &P, const P3d &Q, // V3d PQ
            Real len_of_arrow = -1,
            Real rad_of_arrow = -1,
            int order = 25,
            ThinOrThick style = THIN,
            Boolean two_arrows = false );
```

Examples of the use of *Arrow3d*, RoundArrow(...), and DrawArrow3d(...) are given in the following listing (compare also Figure 6.11).

Sample code for better understanding:

```
Arrow3d Arrow1, Arrow2, BentArrow1, BentArrow2;

void Scene::Init( )
{
    Arrow1.Def( -1 ); // type 1, from (-1,0,0) to (0,0,0)
    Arrow1.Scale( 3 ); // magnification
    Arrow2.Def( 2 ); // type 2, from (+1,0,0) to (0,0,0)
    Arrow2.Scale( 3, 5, 1 ); // different scalings
    Arrow2.Rotate( Zaxis, 90 );
    Arrow2.Translate( 0, 5, 0 );
    Arrow2.Twist( 90 ); // rotation about "local axis"
    BentArrow1.Def( 3 ); // type 3, positive rotation
    BentArrow1.Scale( 6 );
    BentArrow1.Translate( 0, 0, -3 );
    BentArrow2.Def( -4 ); // type 4, negative rotation
    BentArrow2.Scale( 3 );
    BentArrow2.Translate( 0, 0, 4 );
}

void Scene::Draw( )
{
    ShowAxes( Gray, 8 );
    DrawArrowV3d( Orange, Origin, P3d( 0, 0, 10 ), -1, -1, 30, THICK );
    Arrow1.Shade( Red );
    Arrow2.Shade( Blue );
    BentArrow1.Shade( Green );
    BentArrow2.Shade( Yellow );
    RoundArrow( Green, Zaxis, 3, 1, 90, 270, 5 );
}
```

class *BezierCurve3d*; → declaration in `"bezier.h"`

Describes an integral Bézier curve in 3D by its control polygon. The curve points are computed by means of DeCasteljau's algorithm. We provide methods for manipulating the curve (changing of control points, degree elevation, splitting). The base class is *ParamCurve3d*. Some methods are overwritten.

Constructors:

```
BezierCurve3d( ) { B = C = NULL; }
// B and C are dynamically allocated arrays of control points
```

Definition:

```
void Def( Color c, int total_size,
```

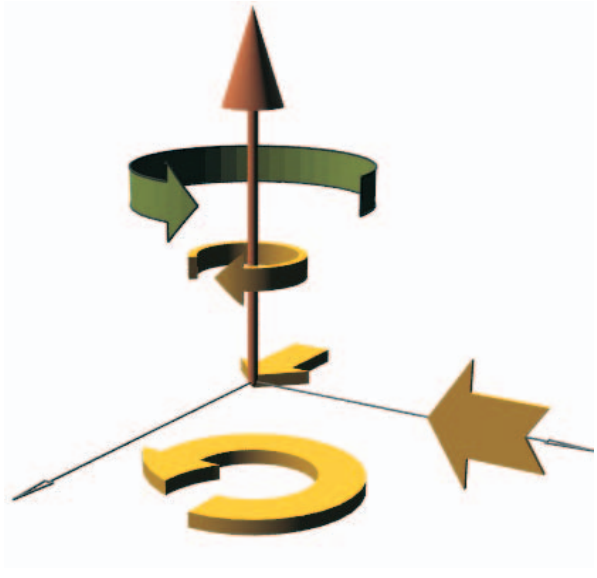

FIGURE 6.11. Different arrows in 3D.

```
            int size_of_control_poly, P3d Control[] );
    // total_size is the number of curve points that will
    // be computed while size_of_control_poly is the number
    // of control points to be used.
```

Frequently used methods and operators:

```
virtual P3d CurvePoint( Real u ) // const
    // Calculate curve point by means of the DeCasteljau algorithm.
virtual StrL3d Tangent( Real u )/* const = 0 */;
virtual Plane OsculatingPlane( Real u )/* const = 0 */;
void DrawControlPolygon( Color col, ThinOrThick thickness );
void MarkControlPoints( Color col, Real rad1, Real rad0 );
void SetControlPoint( int i, P3d P );
    // Redefine control point with index i.
P3d GetControlPoint( int i ) { return B[i]; };
    // Return control point with index i.
int GetPNum( ) { return PNum; }
    // PNum is the number of control points.
void ElevateDegree( int i ); // Elevate the degree by i.
void ReversePolygon( void );
    // Reverse the order of the control points
    // (useful after splitting of the curve).
void Split( Real u, Boolean second_half );
    // Split the Bézier curve at the parameter value u.
    // Depending on the value of second_half the new
    // curve is identical to the part of the
```

```
// old curve that corresponds to the parameter
// interval [0, u] or [u, 1].
```

Sample code for better understanding:

```
BezierCurve3d BezierCurve;
P3d P[3];
P[0]( 0, 0, 0 ), P[1]( 4, 0, 4 ), P[2]( 0, 4, 8 );
BezierCurve.Def( Black, 200, 3, P );
BezierCurve.ElevateDegree( 2 );
BezierCurve.Split( 0.5 );
BezierCurve.Draw( VERY_THICK );
BezierCurve.DrawControlPolygon( Blue, THICK );
BezierCurve.MarkControlPoints( Blue, 0.15, 0.1 );
```

class *BezierSurface*; → declaration in `"bezier.h"`

Describes an integral Bézier surface by its control polygon. The surface points are computed by means of DeCasteljau's algorithm. We provide methods for manipulating the surface (changing of control points, degree elevation, splitting). The base class is *ParamSurface*.

Constructors:

```
BezierSurface( ) { B = C = NULL; }
// B and C are dynamically allocated 2D arrays of control points.
// C is needed for the DeCasteljau algorithm.
```

Definition:

```
void Def( Color c, int m, int n, int pnum1, int pnum2,
      P3d **Control );
// m and n determine the number of facets in the u and v directions.
// pnum1 and pnum2 are the dimensions of the control net.
```

Frequently used methods and operators:

```
P3d SurfacePoint( Real u, Real v ) { return DeCasteljau( u, v ); }
void DrawControlPolygon( Color col, ThinOrThick style,
      Real offset = STD_OFFSET );
void MarkControlPoints( Color col, Real rad1, Real rad0 = 0 );
P3d GetControlPoint( int i, int j ) { return B[i][j]; };
void SetControlPoint( int i, int j, P3d P );
   // Redefine control point with indices i and j.
int GetPNum1( ) { return PNum1; };
int GetPNum2( ) { return PNum2; };
   // Two methods that return the dimension of the control net.
void ElevateDegree( int m, int n );
void ReversePolygon( Boolean u_dir, Boolean v_dir );
```

```
    // Reverse the order of the control net in u and/or v
    // direction according to the value of the Boolean arguments.
  void SplitU( Real u, Boolean second_half );
    // Split the Bézier surface at the parameter value u.
    // Depending on the value of second_half the new
    // curve is identical to the part of the old curve that
    // corresponds to the parameter interval [0,u] or [u,1],
    // respectively.
  void SplitV( Real v, Boolean second_half );
    // Analogous to SplitU(...).
```

Sample code for better understanding:

```
BezierSurface BezSurf;
P3d **P = NULL;

ALLOC_2D_ARRAY( P3d, P, 3, 3, "P" );

P[0][0].Def( -8, -8, -4 );
P[0][1].Def( -8,  0,  4 );
P[0][2].Def( -8,  8, -4 );
P[1][0].Def( -2, -8,  5 );
P[1][1].Def( -2,  0,  8 );
P[1][2].Def( -2,  8,  5 );
P[2][0].Def(  2, -8,  5 );
P[2][1].Def(  2,  0,  8 );
P[2][2].Def(  2,  8,  5 );

BezSurf.Def( Blue, 31, 61, m, n,P );
BezSurf.SplitU( 0.5, true );
BezSurf.SplitV( 0.5, false );
BezSurf.ElevateDegree( 1, 2 );
BezSurf.SetControlPoint( 0, 0, P3d( -12, -8, 3 ) );

FREE_2D_ARRAY( P3d, P, m, n, "P" );

BezSurf.Shade( SMOOTH, REFLECTING );
```

class *Box*; → declaration in `"box.h"`

Describes an ordinary box in 3-space. Inherits methods and operators from *O23d* and *O3d*.

Constructors:

```
Box( ); // Default constructor.
Box( Color col, Real x, Real y, Real z );
    // x, y, and z are the box dimensions. In the initial position,
    // the three box sides lie on the positive coordinate axes;
    // one box corner is at the origin.
```

Operators:

```
void operator = ( const Box &b ); // Copy operator.
```

Definition:

```
void Def( Color col, Real x, Real y, Real z ); // Compare constructor.
void Def( Color col, const P3d &min, const P3d &max );
    // Define box through opposite vertices. In the initial position,
    // the box sides are parallel to the coordinate axes.
```

Additional methods:

```
void WireFrame( Boolean remove_hidden_lines,
        ThinOrThick thick, Real offset = STD_OFFSET );
    // Draw the edges (wire frame).
void Shade( Boolean reflect = true );
    // Shade reflecting or matte.
void ShadeBackfaces( Boolean reflect = true );
    // Shade only the backfaces (reflecting or matte).
    // This is useful for transparent boxes.
void Transform( Polyhedron &P );
void TransformIntoPolyhedron( );
    // Two conversion methods for transformation into a polyhedron.
void GetSides( Real &x, Real &y, Real &z ) const; // Side lengths.
P3d GetCenter( ); // Return box center.
Real GetDiameter( ); // Return box diameter.
```

class *Cad3Data*; → declaration in `"solid.h"`

This class is important for the user (not the above class *ArrayOfSolids*!). Describes an arbitrary cluster of solids generated by means of CAD3D. Whole scenes — even several scenes — can be read from `"*.llx"` and `"*.llz"` data files, manipulated (translated, rotated, scaled), smooth-shaded and finally exported to POV-Ray.

Constructors:

```
Cad3Data( int size = 100 );
```

Definition:

```
void Def( int size );
  // Maximum number of scenes to be read
  // (each scene may still consist of many solids).
void ReadNewMember( const char *name, Real scale_factor = 0.05 );
  // Read new solid or group of solids and scale it at the
  // same time. Usually, CAD3D scenes are much too
  // large for the optimized OPEN GEOMETRY range.
  // Thus, they are multiplied by a comparatively small factor.
```

Operators:

```
ArrayOfSolids & operator [ ] ( int i );
  // Return the i-th "scene" (one or several solids);
  // the k-th solid of this array is called by [i][k].
```

Additional methods:

```
void Shade( FaceMode fmode = ONLY_FRONTFACES );
  // Display everything smooth-shaded;
  // fmode = true speeds up the display.
void WireFrame( Boolean remove_hidden_lines, ThinOrThick thick,
                                        Real offset = 1e-3 );
  // remove_hidden_lines plots the solids first and then
  // adds the wire frame (kind of fast hidden line removal).
void ReadNewMember( const char *name, Real scale_factor = 0.05 );
  // Read single solids ("*.llx") or whole scenes ("*.llz").
void DeleteMember( int i );
  // Remove the corresponding scene or solid.
int Size( );
  // Return number of members.
void MakeInvisible( int i );
  // After this command, the i-th member will not be drawn
  // (although it is not removed!).
void MakeVisible( int i );
  // Draw again.
void GetBoundingBox( Box &box );
P3d GetCenterOfBoundingBox( );
void ShowBoundingBox( );
void OptimizeView( );
  // Fit everything perfectly into an OPEN GEOMETRY environment.
void SetAllColors( Color c );
  // Redefine color for all the members.
void SetRandomColors( );
  // Show the different members in different (random) colors.

// Geometric transformations:
void Translate( Real dx, Real dy, Real dz );
void Translate( const V3d &t );
void Rotate( const StrL3d &axis, Real angle_in_deg );
void Rotate( const RotMatrix &R );
void ExportTo_POV_Ray( const char *name );
  // For high-quality output.
void Delete( );
```

FIGURE 6.12. Output of the sample file for *Cad3Data*.

Sample code for better understanding:

```
Cad3Data Data;

void Scene::Init( )
{
    Data.ReadNewMember( "DATA/LLZ/enterprise.llz", 0.08 );
    AllowRestart( );
}
void Scene::Draw( )
{
    if ( FrameNum( ) % 2 )
        Data.Shade( );
    else
        Data.WireFrame( true, MEDIUM );
}
void Scene::Animate( )
{
}
void Scene::CleanUp( )
{
}
void Projection::Def( )
{
    if ( VeryFirstTime( ) )
    {
        DefaultCamera( 18, 18, 12 );
        Data.OptimizeView( );
    }
}
```

class *Circ3d*; → declaration in `"circ3d.h"`

Describes a circle in 3D. Derived from *O3d*, *Poly3d*, and *RegPoly3d* (many methods inherited).

Constructors:

```
Circ3d( ) // Default constructor.
Circ3d( Color col, const P3d &Center, const V3d &Normal,
        Real rad, int n, FilledOrNot filled = EMPTY )
   // Classical definition via center, axis direction, radius.
Circ3d( Color col, const P3d &FirstElem, const StrL3d &a,
        int n, FilledOrNot filled = EMPTY )
   // Definition via point on circle and axis.
Circ3d( Color col, const P3d &A, const P3d &B, const P3d &C,
        int n, FilledOrNot filled = EMPTY )
   // Circle is given by three points.
```

Definition:

```
void Def( Color col, const P3d &Center, const V3d &Normal,
          Real rad, int n, FilledOrNot filled = EMPTY )
   // Classical definition via center, axis direction, radius.
void Def( Color col, const P3d &FirstElem, const StrL3d &a, int n,
                     FilledOrNot filled = EMPTY )
   // Definition via point on circle and axis.
void Def( Color col, const P3d &A, const P3d &B, const P3d &C,
                     int n, FilledOrNot filled = EMPTY );
   // Circle is given by three points.
```

Additional methods:

```
int SectionWithPlane( const Plane &e, P3d &S1, P3d &S2 );
   // Returns the number of intersection points with a plane and
   // stores them in S1 and S2.
P3d P( Real t );
   // Returns a point on the parameterized circle (t in arc length).
```

class *ComplexPoly3d*; → declaration in `"complex_poly.h"`

Closed non convex polygon with holes (loops) in 3-space. Derived from *Poly3d*.

Constructors:

```
ComplexPoly3d( ); // Default constructor.
```

Definition:

```
void Def( Color f, int NLoops, Poly3d *loop, int oriented = 0 );
void Def( Color f, int NLoops, Poly2d *loop, int oriented = 0 );
   // NLoops is the number of loops, loop is an array of polygons,
   // i.e., a pointer to the first polygon.
void Def( ComplexPoly3d *cp ); // Copy complex polygon cp.
```

```
void Def( ComplexPoly2d *cp ); // Copy complex polygon cp.
```

Additional methods:

```
void Translate( V3d t );
void Rotate( const StrL3d &a, Real angle_in_deg );
void Scale( Real kx, Real ky, Real kz );

void Shade( Shininess reflect = REFLECTING );
   // Does not work in case of intersecting loops!
void Outline( Color col, ThinOrThick style,
              Real offset = STD_OFFSET ); // Draw the outline.
```

class *Conic3d*; → declaration in `"conic.h"`

Describes a conic in 3-space. Derived from the 2D class *Conic*. Computations will be performed by projecting everything onto one of the coordinate planes.

Constructors:

```
Conic3d( ) { proj_dir = 1; } // Default constructor.
```

Definition:

```
Conic3d & operator = ( Conic3d &other );
void Def( Color col, int numPoints, P3d P[5] );
// numPoints is the number of points on the approximating
// polygon. Calculations with the conic are not done with these
// approximations but with the exact mathematical equation!
// The conic is defined by five points.
void Def( Color col, int numPoints, const StrL3d t[5] );
// Definition through five tangents.
void Def( Color col, int numPoints, const P3d P[3],
          const P3d &T, const StrL3d &t );
// Definition through three points and point plus tangent.
void Def( Color col, int numPoints, const P3d &M,
          const P3d &A, const P3d &B );
// Definition through pair of conjugate diameters.
void Def( Color col, int numPoints, const P3d &P,
          const P3d &F1, const P3d &F2, TypeOfConic type );
// Definition through point plus two focal points and type of conic;
// type can be either ELLIPSE or HYPERBOLA.
```

Frequently used methods and operators:

```
P3d GetCenter( ) { return T2( M ); } // Return the center.
P3d GetA( ) const { return T2( A ); }
P3d GetB( ) const { return T2( B ); }
   // A and B are points on the major axis.
P3d GetC( ) const { return T2( C ); }
P3d GetD( ) const { return T2( D ); }
```

```
    // C and D are points on the minor axis.
Real3d GetM( ) const { return T2( M ); } // returns the center
Real DistMA( ) const { return (GetA( ) - GetM( )).Length( ); }
Real DistMC( ) const { return (GetC( ) - GetM( )).Length( ); }
StrL3d MajorAxis( ) const { return T3( Axis1 ); }
StrL3d MinorAxis( ) const { return T3( Axis2 ); }
StrL3d Asymptote1( ) const { return T3( As1 ); }
StrL3d Asymptote2( ) const { return T3( As2 ); }
int SectionWithPlane( const Plane s, P3d &S1, P3d &S2 );
    // Return the number of intersection points (0, 1, or 2)
    // and calculates these points.
Plane GetPlane( ) const { return p; } // Returns the carrier plane.
void Draw( ThinOrThick thick, Real offset = STD_OFFSET,
      Real max_radius = DUMMY ) /* const */;
void AssignTo( ComplexL3d &c );
    // Assign conic to object of type ComplexL3d.
StrL3d GetPolar( const P3d &Pole ) const;
    // Return polar of point P.
P3d GetPole( const StrL3d &polar ) const;
    // Return pole of straight line polar.
```

class *DiffEquation3d*; → declaration in `"diff_equation.h"`

Describes a differential equation (3D vector field) and provides a solving method (RUNGE–KUTTA). Abstract class due to the purely virtual member function TangentVector(...). In order to use it, you must derive a class of your own from *DiffEquation*. It is derived from *L3d*. Thus, you can immediately plot the solution curve. Requires the inclusion of `"diff_equation.h"` at the beginning of your file. For a sample code listing, please refer to the corresponding 2D class *DiffEquation*.

Constructor:

```
DiffEquation3d( ); // Default constructor.
```

Additional methods:

```
virtual void Solve( Real t1, Real t2, Real h, const P3d &StartPoint );
    // Solves differential equation by means of RUNGE−KUTTA.
virtual V3d TangentVector( Real t, const P3d &P ) = 0;
    // Virtual function to describe the corresponding 3D vector field.
    // Has to be implemented by the user!
```

class *FunctionGraph*; → declaration in `"paramsurface.h"`

Abstract class derived from *ParamSurface*. Describes a function graph over a parameter rectangle. In order to use it, you have to implement the purely virtual member function z(*Real* x, *Real* y).

Methods:

```
virtual Real z( Real x, Real y ) = 0;
P3d SurfacePoint( Real u, Real v ); // Return P3d ( u, v, z( u, v ) ).
V3d Normal( Real x, Real y ); // Return normalized normal.
void ShadeSolidBlockUnderGraph( Color c, Real zmin,
        ThinOrThick borderLines = MEDIUM );
   // Shade the block under the function graph.
void TransformIntoPolyhedron( );
   // Transform function graph into polyhedron (useful if you need
   // methods of the class Polyhedron).

// Partial derivates of first order:
Real fu( Real u, Real v );
Real fv( Real u, Real v );

// Partial derivates of second order:
Real fuu( Real u, Real v );
Real fuv( Real u, Real v );
Real fvu( Real u, Real v );
Real fvv( Real u, Real v );

virtual int GetIndicatrix( Conic3d &conic, Real u, Real v,
        Real scale = 1, int num_of_points = 120 );
   // Compute DUPIN indicatrix and store it in conic.
```

Sample code for better understanding:

```
#include "opengeom.h"
#include "defaults3d.h"

class TheSurface: public FunctionGraph
{
public:
   virtual Real z( Real x, Real y )
   {
      x *= 0.3; y *= 0.3;
      x += 0.4* y;
      return  0.2 * x * x + 0.5 * y * y;
   }
};
TheSurface F;
void Scene::Init( )
{
   F.Def( LightYellow, 30, 30, -12, 12, -7, 7 );
   F.PrepareContour( );
}

void Scene::Draw( )
```

```
{
   F.WireFrame( Black, 6, 6, MEDIUM );
   F.Contour( Black, MEDIUM );
   F.Shade( SMOOTH, REFLECTING );
   F.ShadeSolidBlockUnderGraph( LightGray, 0 );
}
```

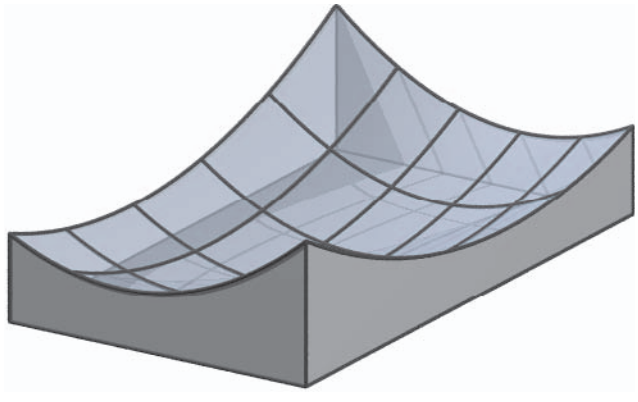

FIGURE 6.13. A function graph (compare sample code).

class *HelicalSurface*; → declaration in `"revol.h"`

Describes a helical surface with axis z. Derived from *ParamSurface*.

Definition:

```
void Def( Color c, Real param, int rot_number, Spline3d &k,
          Real w1, Real w2 );
   // The surface is swept by the spline curve k that undergoes
   // a helical motion with parameter param and axis z.
   // w1 and w2 are the parameter limits (screw angles in degrees).
```

Sample code for better understanding:

```
#include "opengeom.h"
#include "defaults3d.h"

HelicalSurface Surf;

void Scene::Init( )
{
   Coord3dArray points = // some points on the generating line
   {
```

```
        { 0, 5, 0 },{ 0, 3, 2 }, { 0, 1, 0 }, { 0, 3, −2 }
    };
    CubicSpline3d m;
    m.Def( Red, 4, points, 10 );
    const Real H = 6, parameter = H / ( 2 * PI );
    Surf.Def( Green, parameter, 200, m, −360, 270 );
    Surf.PrepareContour( );
}
void Scene::Draw( )
{
    Zaxis.Draw( Black, −8, 7, THICK );
    Surf.Shade( SMOOTH, REFLECTING );
    Surf.DrawBorderLines( Black, MEDIUM );
    Surf.Contour( Black, MEDIUM );
}
```

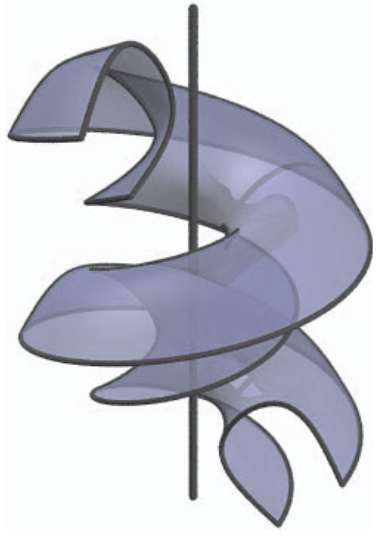

FIGURE 6.14. A helical surface (compare sample code).

class *L3d*; → declaration in `"lines3d.h"`

Describes a line (arbitrary polygon) in 3D. Derived from *O3d*. Equipped with a number of very powerful methods. Base class for other frequently used classes (*ParamCurve3d*, *PathCurve3d*).

Constructors:

L3d( ) // Default constructor.
L3d(*Color* f, **int** numOfPoints)

```
    // Specify color and number of points.
  L3d( Color f, int N, Coord3dArray P)
    // Specify color coordinates of points.
```

Drawing methods:

```
  void Draw( ThinOrThick thick, Real offset = STD_OFFSET,
             Real max_dist_of_2_points = -1 );
    // Standard drawing method. For max_dist_of_2_points > 0 a line
    // segment is only drawn, when the two adjacent points have a
    // smaller distance. This is useful for the drawing of curves with
    // points at infinity.
  void LineDotted( int n, ThinOrThick thick,
             Real offset = STD_OFFSET );
    // Draw curve line-dotted with n segments.
  void DrawPartInsideCircle( const P3d &M, Real rad,
             ThinOrThick style, Real offset );
    // Draw only part inside circle with center M and radius rad.
  void Draw2d( ThinOrThick thick);
    // Draw the curve in 2D only (ignoring z-coordinates).
  void DrawUprightProjCylinder( Real z_min, Color lineColor,
          ThinOrThick lineStyle, Color surfColor, Boolean shade );
    // Method for drawing and shading the projection cylinder
    // (cylinder through line with z-parallel rulings).
```

Methods:

```
  void RotateIntoPlane( const Plane &e, const StrL3d &a,
             Boolean curve_is_plane = false );
    // Rotate all element points about the axis a into the plane e.
  Plane CarrierPlane( );
    // Return the carrier plane of the line (only for plane lines).
  Boolean TangentsFromPoint( P3d P, L3d &k );
    // Try to find tangents of the line through P. The points of
    // tangency are stored in k (only if P and all line elements lie
    // in a common plane).
  Plane OsculatingPlane( int i );
    // Approximate the osculating plane in point with index i.
  V3d ApproxTangentVector( int i );
    // Approximate the tangent vector in point with index i.
  StrL3d ApproxTangent( int i );
    // Approximate the tangent in point with index i.
  Boolean isClosed( Real tol = 0 ) // Checks whether line is closed
  Real Length( ); // Return length of line (polygon).
  Real ArcLength( int i1, int i2 );
    // Return length of line between points with indices  i1 and i2.
  void SmoothTo( int n, Spline3d &k );
    // Compute a spline with n points that interpolates the line.
  void DrawPartInsideCircle( const P3d &M, Real rad,
             ThinOrThick style, Real offset );
    // Draw part inside circle with center M and radius rad.
  int SectionWithPlane( const Plane &e, O3d &S );
    // Compute the intersection points of the line with the plane e
    // and stores them in the O3d object S.
```

Sample code for better understanding:

```
// Compare "smooth_spline.cpp" and Figure 6.15.
// Note that we deal with 3D objects but draw in 2D!
L3d Line;
CubicSpline3d Spline;
Coord3dArray P = { { -1, 0, 0 }, { 0, -1.41, 0 },
    { 2, 0, 0 }, { 0, 2.82, 0 }, { -4, 0, 0 }, { 0, -5.64, 0 } };
Line.Def( Black, 6, P );
Line.SmoothTo( 100, Spline );
Spline.ChangeColor( Red );
ShowAxes2d( Gray, 7, 6 );
Line.Draw( MEDIUM );
Spline.Draw( THICK );
```

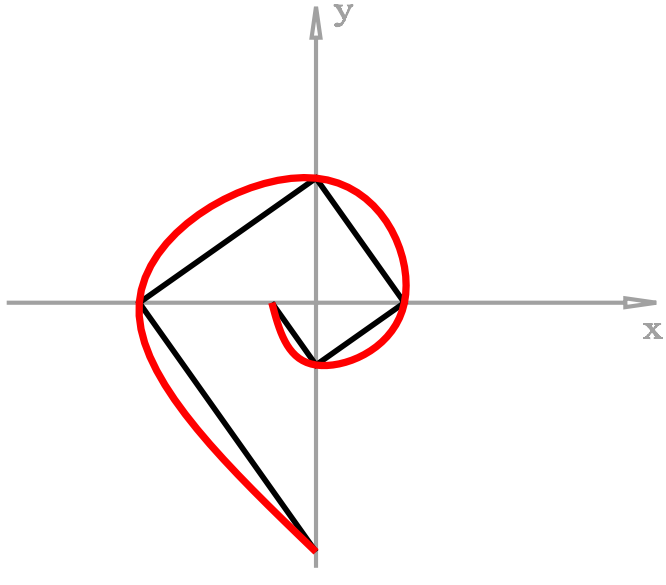

FIGURE 6.15. A 2D polygon (object of type *L3d*) is interpolated by a spline curve.

class *NUBS3d*; → declaration in `"nurbs.h"`

Describes an integral B-spline curve in 3D by a control polygon and a knot vector. The curve points are computed by means of the Cox–De Boor algorithm. We provide methods for defining and displaying the curve. Dynamic memory allocation is handled internally. The base class is *ParamCurve3d*.

Constructors:

NUBS3d( ) { T = A = NULL; D = E = NULL; }
// Set all pointers to NULL. T is the knot vector, D the control
// polygon. A and E are used for internal computations.

Definition:

void Def(*Color* c, **int** size, **int** knum, *Real* K[], **int** pnum, *P3d* P[]);
void Def(*Color* c, **int** size, **int** pnum, *P2d* P[], **int** continuity,
Boolean closed = false);

The first defining method allows the user to set the knot vector as well as the control points according to her/his wishes. The second method sets the knot vectors automatically in order to ensure C^k-continuity (k = continuity). Furthermore, the user can choose between open and closed B-splines.

Frequently used methods and operators:

virtual *P3d* CurvePoint(*Real* u) { **return** DeBoor(u); }
// Overwrite the CurvePoint(...) function of *ParamCurve3d*.
void MarkControlPoints(*Color* c, *Real* rad1, *Real* rad0 = 0);
void DrawControlPolygon(*Color* c, *ThinOrThick* style);
Real GetKnot(**int** i) {**return** T[i]; };
P2d GetControlPoint(**int** i) { **return** D[i]; };
int GetKnotNum( ) { **return** M; }
int GetPointNum( ) { **return** N; }

Sample code for better understanding:

```
NUBS3d Spline;

const int number_of_control_points = 5;

P3d Point[number_of_control_points];
Point[0].Def( -10, 0, 0 );
Point[1].Def( -6, 6, 5 );
Point[2].Def( -2, 0, 0 );
Point[3].Def( -2, -6, 5 );
Point[4].Def( -5, -6, 0 );

const int total_size = 49;
const int continuity_class = 2;

// create a closed spline curve
Spline.Def( Black, total_size, number_of_control_points,
    Point, continuity_class, true );

Spline.Draw( THICK );
Spline.DrawControlPolygon( Blue, THIN );
Spline.MarkControlPoints( Blue, 0.2, 0.1 );
```

class *NUBS_Surf*; → declaration in `"nurbs.h"`

Describes an integral B-spline surface by a control net and two knot vectors. The curve points are computed by means of the Cox–De Boor algorithm. We provide methods for defining and displaying the surface. Dynamic memory allocation is handled internally. The base class is *ParamSurface*.

Constructors:

```
NUBS_Surf( ) { T1 = T2 = A1 = A2 = NULL; D = NULL;
                                  E1 = E2 = NULL; }
  // Set all pointers to NULL. T1 and T2 are the knot vectors.
  // D is the control net. The remaining pointers are used for
  // internal computations.
```

Definition:

```
void Def( Color c, int size1, int size2, int knum1, Real K1[],
   int knum2, Real K2[], int pnum1, int pnum2, P3d **P );
void Def( Color c, int size1, int size2, int pnum1, int pnum2,
   P3d **P, int continuity1, int continuity2,
   Boolean closed1 = false, Boolean closed2 = false );
```

The first defining method allows the user to set the knot vectors as well as the control net according to her/his wishes. The second method sets the knot vectors automatically in order to ensure C^{k_i}-continuity (k_1 = continuity_1, k_2 = continuity_2) of the u- and v-parameter lines, respectively. Furthermore, the user can choose between open and closed B-splines in both parameter directions.

Frequently used methods and operators:

```
virtual P3d SurfacePoint( Real u, Real v ) { return DeBoor( u, v ); }
   // Overwrite the SurfacePoint(...) function of ParamSurface.
void MarkControlPoints( Color c, Real rad1, Real rad0 = 0 );
void DrawControlPolygon( Color c, ThinOrThick style );
Real GetKnot1( int i ) { return T1[i]; };
Real GetKnot2( int i ) { return T2[i]; };
P3d GetControlPoint( int i, int j ) { return D[i][j]; };
int GetKnotNum1( ) { return M1; }
int GetKnotNum2( ) { return M2; }
int GetPointNum1( ) { return N1; }
int GetPointNum2( ) { return N2; }
```

Sample code for better understanding:

```
NUBS_Surf Surf;
int pnum1 = 5, pnum2 = 7; // dimension of control nets
P3d **P = NULL;
ALLOC_2D_ARRAY( P3d, P, pnum1, pnum2, "P" );
int i, j;
for ( i = 0; i < pnum1; i++ )
    for ( j = 0; j < pnum2; j++ )
    {
        P[i][j].Def( 3, 0, 0 );
        P[i][j].Rotate( Yaxis, i * 360.0 / pnum1 );
        P[i][j].Translate( 10, 0, 0 );
        P[i][j].Rotate( Zaxis, j * 360.0 / pnum2 );
    }
// create a spline surface that is closed in one parameter direction
Boolean close_u = true;
Boolean close_v = false;
Surf.Def( Yellow, 50, 50, pnum1, pnum2, P, 3, 3, close_u, close_v );
FREE_2D_ARRAY( P3d, P, pnum1, pnum2, "P" );

Surf.Shade( SMOOTH, REFLECTING );
Surf.DrawControlPolygon( Black, MEDIUM );
Surf.MarkControlPoints( Black, 0.25, 0.15 );
```

class *NURBS3d*; → declaration in `"nurbs.h"`

Describes a rational B-spline curve in 3D by a control polygon, an array of weights and a knot vector. The curve points are computed by applying the Cox–De Boor algorithm in 4D. We provide methods for defining and displaying the curve. Dynamic memory allocation is handled internally. The base class is *ParamCurve3d*.

Constructors:

```
NURBS3d( ) { T = A = W = X = NULL; D = E = NULL; }
// Set all pointers to NULL. T is the knot vector, D the control
// polygon. A and E are used for internal computations.
```

Definition:

```
void Def( Color c, int size, int knum, Real K[], int pnum, P2d P[],
    Real weight[] );
void Def( Color c, int size, int pnum, P2d P[], Real weight[],
    int continuity, Boolean closed = false );
```

The first defining method allows the user to set the knot vector as well as the control points according to her/his wishes. The second method

sets the knot vectors automatically in order to ensure C^k-continuity (k = continuity). Furthermore, the user can choose between open and closed B-splines.

Frequently used methods and operators:

```
virtual P2d CurvePoint( Real u ) { return DeBoor( u ); }
    // Overwrite the CurvePoint(...) function of ParamCurve3d.
void MarkControlPoints( Color c, Real rad1, Real rad0 = 0 );
void DrawControlPolygon( Color c, ThinOrThick style );
Real GetKnot( int i ) {return T[i]; };
P2d GetControlPoint( int i ) { return D[i]; };
Real GetWeight( int i ) { return W[i]; };
int GetKnotNum( ) { return M; }
int GetPointNum( ) { return N; }
```

Sample code for better understanding:

```
NURBS3d Spline;

const int number_of_control_points = 5;

P3d Point[number_of_control_points];
Point[0].Def( -10, 0, 0 );
Point[1].Def( -6, 6, 5 );
Point[2].Def( -2, 0, 0 );
Point[3].Def( -2, -6, 5 );
Point[4].Def( -5, -6, 0 );

Real weight[number_of_control_points];
int i;
for ( i = 0; i < number_of_control_points; i++ )
    weight[i] = 1;
weight[3] = 3;

const int total_size = 49;
const int continuity_class = 3;

Spline.Def( Black, total_size, number_of_control_points,
    Point, weight, continuity_class );
Spline.Draw( THICK );
Spline.DrawControlPolygon( Blue, THIN );
Spline.MarkControlPoints( Blue, 0.2, 0.1 );
```

class *NURBS_Surf*; → declaration in `"nurbs.h"`

Describes a rational B-spline surface by a control net, a 2D array of weights and two knot vectors. The curve points are computed by means of the Cox–De Boor algorithm. We provide methods for defining and displaying the surface. Dynamic memory allocation is handled internally. The base class is *ParamSurface*.

Constructors:

```
NURBS_Surf( ) { T1 = T2 = A1 = A2 = NULL; W = NULL;
  D = NULL; E1 = E2 = NULL; }
// Set all pointers to NULL. T1 and T2 are the knot vectors, D is the
// control net, W the knot vector. The remaining pointers are used
// for internal computations.
```

Definition:

```
void Def( Color c, int size1, int size2, int knum1, Real K1[],
   int knum2, Real K2[], int pnum1, int pnum2,
   P3d **P, Real **Weight );
void Def( Color c, int size1, int size2, int pnum1, int pnum2,
   P3d **P, Real **Weight, int continuity1, int continuity2,
   Boolean closed1 = false, Boolean closed2 = false );
```

The first defining method allows the user to set the knot vectors as well as the control net and the weights according to her/his wishes. The second method sets the knot vectors automatically in order to ensure C^{k_i}-continuity (k_1 = continuity1, k_2 = continuity2) of the u- and v-parameter lines, respectively. Furthermore, the user can choose between open and closed B-splines in both parameter directions.

Frequently used methods and operators:

```
virtual P3d SurfacePoint( Real u, Real v ) { return DeBoor( u, v ); }
   // Overwrite the SurfacePoint function of ParamSurface.
void MarkControlPoints( Color c, Real rad1, Real rad0 = 0 );
void DrawControlPolygon( Color c, ThinOrThick style );
Real GetKnot1( int i ) { return T1[i]; };
Real GetKnot2( int i ) { return T2[i]; };
P3d GetControlPoint( int i, int j ) { return D[i][j]; };
int GetKnotNum1( ) { return M1; }
int GetKnotNum2( ) { return M2; }
int GetPointNum1( ) { return N1; }
int GetPointNum2( ) { return N2; }
```

Sample code for better understanding:

```
NURBS_Surf Surf;
int pnum1 = 5, pnum2 = 7; // dimension of control nets
P3d **P = NULL;
Real **Weight = NULL;
ALLOC_2D_ARRAY( P3d, P, pnum1, pnum2, "P" );
ALLOC_2D_ARRAY( Real, Weight, pnum1, pnum2, "Weight" );
int i, j;
for ( i = 0; i < pnum1; i++ )
    for ( j = 0; j < pnum2; j++ )
    {
        P[i][j].Def( 3, 0, 0 );
        P[i][j].Rotate( Yaxis, i * 360.0 / pnum1 );
        P[i][j].Translate( 10, 0, 0 );
        P[i][j].Rotate( Zaxis, j * 360.0 / pnum2 );
        Weight[i][j] = 1;
    }
Weight[4][4] =  10;
// create a spline surface that is closed in one parameter direction
Boolean close_u = true;
Boolean close_v = false;
Surf.Def( Yellow, 50, 50, pnum1, pnum2, P, Weight,
    3, 3, close_u, close_v );
FREE_2D_ARRAY( P3d, P, pnum1, pnum2, "P" );

Surf.Shade( SMOOTH, REFLECTING );
Surf.DrawControlPolygon( Black, MEDIUM );
Surf.MarkControlPoints( Black, 0.25, 0.15 );
Surf.GetControlPoint( 4, 4 ).Mark( Red, 0.3 );
```

class *O3d*; → declaration in `"o3d.h"`

Describes a conglomerate of points in 3-space. Derived from *O23d* and *Prop3d*.

Constructors:

O3d( ) // Default constructor.
O3d(*Color* f, **int** n) // Specify color and size.

Definition:

void Def(*Color* f, **int** n)
void Def(*Color* f, **int** n, *Coord3dArray* P)
void SameAs(*O3d* *obj);

Geometric transformations (apply to all objects derived from O3d!):

```
void Translate( Real dx, Real dy, Real dz );
void Translate( const V3d &v ) { Translate( v.x, v.y, v.z ); }
void Scale( Real kx, Real ky = 0, Real kz = 0 );
void Rotate( const StrL3d &a, Real w );
void Rotate( const RotMatrix &matrix );
   // Define rotation via matrix.
void Screw( const StrL3d &a, Real angle, Real parameter );
void Screw( const HelicalMotion &S );
void MoveGenerally( Real dx, Real dy, Real dz, Real rot1, Real rot2 );
   // Rotates the object through rot1° about the x-axis, through
   // rot2° about the z-axis and translates it by the vector (dx, dy, dz)
   // (in that order!).
void Reflect( const Plane &e );
void Reflect( const P3d &Center );
void Inversion( const Sphere &sphere );
   // Inverts object on sphere.
void FitIntoSphere( Real radius );
void FitIntoSphere( Real radius, P3d &center, Real &scale_factor );
   // A sometimes useful routine when you import data
   // from other systems like 3D Studio Max or Cad3D.
```

Manipulating the element points:

```
void SetPoint( int i, Real x0, Real y0, Real z0 );
void AssignVertices( Coord3dArray koord );
P3d GetPoint( int i ) const
```

class *O3dGroup*; → declaration in `"groups.h"`

It is possible to combine several objects of type *O3d* into an object group *O3dGroup*. This is convenient for the creation of a scene that is made up from different more or less identical subscenes.

Constructors:

```
O3dGroup( int N = 0 ); // N is the number of elements.
```

Operators:

```
O3d * operator [ ] ( int i );
   // Return an O3d pointer to the i-th element.
```

Methods:

```
void Def( int N ); // Work with N elements.
int N( ) { return n;} // Return the number of elements.

void AddMember( int i, O3d &obj,
   TypeOfObject type = OG_DEFAULT_OBJECT );
    // Add a new object O3d.
void AddChild( int i, O3dGroup &obj ); // Add another object group.
```

```
void Elem( int i, int j, Real x, Real y, Real z );
   // Set the coordinates of the j-th point of the i-th object.

// Geometric transformations.
void Translate( Real dx, Real dy, Real dz );
void Translate( const V3d &v ) { Translate( v.x, v.y, v.z ); }
void Scale( Real kx, Real ky, Real kz );
void Scale( Real k );
void Rotate( const StrL3d &a, Real w );
void Screw( const StrL3d &a, Real angle, Real parameter );
void Screw( const HelicalMotion &S );
```

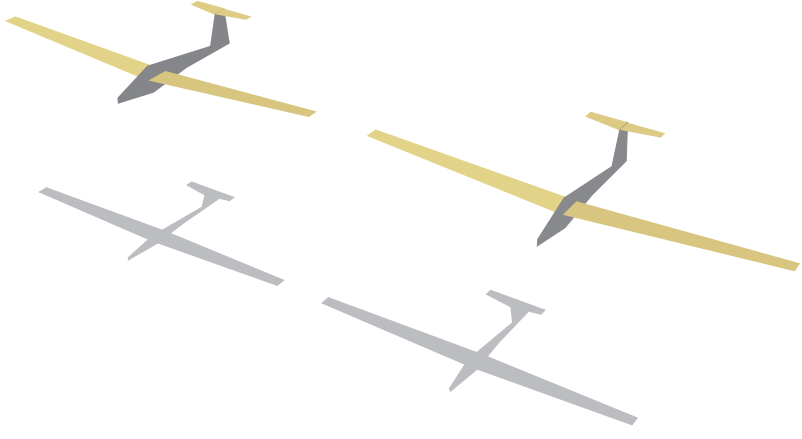

FIGURE 6.16. An example for the use of *O3dGroup* can be found in `"glider.cpp"`: Two gliders are looping the loop and casting shadows on the ground.

class *P3d*; → declaration in `"points.h"`

Describes a 3D point (x, y, z). Inherits the methods and operators of *P2d* and thus also those of *V23d*, *V2d*, *V3d*, and *P23d*.

Constructors:

```
P3d( ); // Default constructor; no initialization.
P3d( Real x0, Real y0, Real z0 ); // Initialization with (x0, y0, z0).
P3d( V3d &v ); // Initialization with a 3D vector.
```

Additional operators:

```
inline friend P3d operator + ( const P3d &v, const V3d &w );
inline friend P3d operator + ( const P3d &v, const P3d &w );
inline friend P3d operator + ( const P3d &v, const V2d &w );
inline friend P3d operator - ( const P3d &v, const V3d &w );
inline friend P3d operator - ( const P3d &v, const P3d &w )
inline friend P3d operator * ( const Real t, const P3d &v );
```

```
// Compatibility with the related classes:
P3d & operator = ( const P3d &P );
P3d & operator = ( const P2d &P );
P3d operator = ( const V3d &v );
```

Additional methods:

```
void Def( Real x0, Real y0, Real z0 );
void Def( Vector v );
   // Two kinds of assigning values to the coordinates.
void Rotate( );
   // Apply a predefined 3D rotation (see Rotation3d)
   // in order to  speed up the code.
void Rotate( const StrL3d &axis, Real angle_in_deg );
   // Rotate the point about an oriented straight line (the axis)
   // through the given angle in degrees.
void Screw( const StrL3d &axis, Real angle_in_deg, Real parameter );
   // Apply a helical motion, given by the axis and parameter.
void Screw( const HelicalMotion &S );
   //  Apply a "screw motion" (helical motion).
void Translate( Real dx, Real dy, Real dz );
   // Translate point by means of the vector (dx, dy, dz).
void Translate( const V3d &v );
   // Translate point by means of the vector v.
void Scale( Real kx, Real ky, Real kz )
   // Apply extended scaling (affine transformation).
P23d Image( ) const;
   // Return the intersection point of the projection
   // ray with "the image plane", i.e., a plane orthogonal
   // to the main projection ray (and coinciding with the
   // target point of the projection).
P3d ReflectedPoint( const Plane &e ) const;
   // Return reflection of the point on the plane.
P3d * ClosestPoint( O3d &obj, int &idx, Real &dmin );
   // Return the point of obj that is closest to the point.
   // Store its index in idx and the minimal distance in dmin.
P3d * ClosestPoint( O3d &obj );
   // Return the point of obj that is closest to the point.
void Reflect( const Plane &e );
   // Reflect the point on the plane.
void Reflect( const P3d &P );
   // Reflect the point on another point.
void MoveGenerally( Real dx, Real dy, Real dz, Real rot1, Real rot2 );
   // Apply a translation by (dx, dy, dz) and two rotations.
   // The first rotation has axis Xaxis and angle of rotation rot1;
   // the second rotation has axis Zaxis and angle of rotation rot2.
Boolean IsInPlane( const Plane &e, Real tol = 1e-4 ) const;
   // Check whether the point coincides with the plane
   // (default tolerance 1e - 4).
Real DistFromPlane( const Plane &e ) const;
```

```
    // Return the oriented distance from the plane.
  Real Dist2( P3d &P ) const;
    // Return the square of the distance to a point.
    // Sometimes, this helps to speed up code, since no
    // sqrt(...) is needed.
  Real Dist_in_xy_projection( P3d &P ) const;
    // Return the distance to another space point in a top view.
  P3d Center( P3d &P ) const;
    // Midpoint between the point and another point P.
  Boolean Collinear( const P3d &P, const P3d &Q,
    Real tol = 1e-4 ) const;
    // Check whether the point coincides with the straight line PQ
    // (default tolerance 1e-4).
  void Mark( Color col, Real r1, Real r2 = 0, int Dim = 3 ) const;
    // Mark a point in space by means of two circles:
    // Outer circle: Radius r1, color col
    // Inner circle: Radius r2 (default is 0), color = background color
    // The circles are interpreted three-dimensionally!
  void MarkPixel( Color col ) const;
    // Plot a pixel at the image of the point (see Image( ) )
  void AttachString( Color col, Real dx, Real dy, char * text ) const;
    // Attach a string to the point (at the "Image" plus a
    // 2D vector (dx, dy) in the "image plane").
  void Inversion( const Sphere &sphere );
    // Invert the point at a given sphere (see class Sphere).
  inline P3d Mid( const P3d &P )
    // Inline version of Center( ).
```

class *ParamCurve3d*; → declaration in `"lines3d.h"`

An *abstract class* for a smooth 3D curve, defined by a parametric representation. Derived from *O3d* and *L3d*. In order to define a specific curve, the user has to derive an additional class from this class!

Definition:

```
  void Def ( Color col, int n, Real umin, Real umax );
  // umin and umax are the parameter limits for the curve.
  // The program will calculate n points on the curve, corresponding
  // to evenly distributed parameter values in between.
```

Additional member variables and methods:

```
  Real u1, u2; // Parameter limits.
  virtual P3d CurvePoint( Real u ) = 0; // dummy
  virtual V3d TangentVector( Real u );
  virtual StrL3d Tangent( Real u );
  virtual void GetOsculatingCircle( Real u, Color col,
              int numPoints, Circ3d &osc_circ );
  // Define the osculating circle at parameter u; it will be
  // approximated by a regular polygon with numPoints points.
  virtual Real ArcLength( Real u1, Real u2, int accuracy = 300 );
  // Return the arc length ∫_{u1}^{u2} sqrt(ẋ² + ẏ² + ż²) du.
  // By default, the curve is interpolated by a 300-sided polygon.
```

Sample code for better understanding:

```
// Compare file "USER/TEMPLETS/paramcurve3d.cpp"!

class MyCurve: public ParamCurve3d
{
public:
    P3d CurvePoint( Real u )
    {
        const Real r = 3, h = r / 6;
        return P3d( r * cos( u ), r * sin( u ), h * Tan( u / 2 ) );
    }
};
MyCurve CubicCircle;
void Scene::Init( )
{
    int number_of_points = 101;
    Real param1 = -3, param2 = -param1;
    CubicCircle.Def( Black, number_of_points, param1, param2 );
}
void Scene::Draw( )
{
    CubicCircle.Draw( THICK );
    Real u0 = 0;
    CubicCircle.Tangent( u0 ).Draw( Black, -5, 5, THIN );
    Circ3d osc_circ;
    CubicCircle.GetOsculatingCircle( u0, PureRed, 200, osc_circ );
    osc_circ.Draw( THIN );
    CubicCircle.CurvePoint( u0 ).Mark( Red, 0.2, 0.1 );
}
```

class *ParamSurface*; → declaration in `"paramsurface.h"`

ParamSurface is one of the most powerful OPEN GEOMETRY classes. It provides a list of methods to perform a number of simple and advanced tasks with surfaces described by parameterized equations. Large parts of Section 3.3 are dedicated to this class. Note that *ParamSurface* is an *abstract class*. That is, in order to use it, you have to derive a class of your own from it. Several special surface classes (*FunctionGraph*, *SurfWithThickness*, *PathSurface*, *SurfOfRevol*, *HelicalSurface*, *RuledSurface*) are derived from *ParamSurface*.

Definition:

```
void Def( Color f, int m1, int m2,
          Real umin, Real umax, Real vmin, Real vmax );
  // m1 is the number of v-lines (u = const.),
```

```
    // m2 is the number of u-lines (v = const.),
    // parameter range: [umin, umax] × [vmin, vmax].
    // You have to implement SurfacePoint(...)!
  void Def( Color f, int m, Real vmin, Real vmax, L3d &k );
    // Sweeping of a 3D curve k (m positions).
    // You have to implement SweepFunction(...)!
```

Public member variables:

```
  int n1, n2; // Number of facets in u- and v-direction.
  Real u1, u2, v1, v2; // Parameter limits.
  StrL3d axis; // Some surfaces have an axis (default: z-axis).
```

Methods for shading and drawing:

```
  void Shade( FlatOrSmooth smooth, Shininess reflecting,
        FaceMode modus = ALL_FACES, Boolean border_lines = false );
    // Principal shading method.
    // Possible values for smooth: FLAT, SMOOTH,
    // possible values for reflecting: MATTE, REFLECTING,
    // VERY_REFLECTING, SUPER_REFLECTING,
    // possible values for modus: ALL_FACES, ONLY_BACKFACES,
    // ONLY_FRONTFACES.

  void ULines( Color c, int n, ThinOrThick style,
          Real offset = STD_OFFSET );
    // Plot approx. n u-lines. The actual number depends on the
    // number n2 of surface facets in u-direction as specified in
    // Def(...). offset can be used to improve z-buffering.
  void VLines( Color c, int n, ThinOrThick style,
          Real offset = STD_OFFSET );
    // Analogous to ULines(...) but in v-direction.
  void GetULine( L3d &uline, const Real v0,
          Color col = Black, int size = 0 );
    // Calculate a specific u-line (v = v0) which is to be passed as
    // first parameter. Color and size can be chosen, but default values
    // are provided. size = 0 will result in the standard number of
    // points in u-direction as specified in Def(...) and returned by
    // NumOfULines( ).
  void GetVLine( L3d &vline, const Real u0,
          Color col = Black, int size = 0 );
    // Compare GetULine(...).
  int NumOfULines( ) // Return the number of u-lines.
  int NumOfVLines( ) // Return the number of v-lines.
  void WireFrame( Color c, int n1 = 0, int n2 = 0,
        ThinOrThick style = THIN, Real offset = STD_OFFSET );
    // Analogous to ULines(...) but in both, u- and v-directions.
  void DrawBorderLines( Color col, ThinOrThick style,
          Boolean b1 = true, Boolean b2 = true,
          Real offset = STD_OFFSET );
    // Draw only the bordering u-lines (if u_lines is true) and the
```

```
    // bordering v-lines (if v_lines is true).
  void Contour( Color f, ThinOrThick style, Real offset = 1e-3 );
    // Draw the outline (silhouette). Requires the calling of
    // PrepareContour( ) in Init( ).
  void PrepareContour( );
    // Prepare the contour. Has to be called only once in Init( ).
```

Methods from differential geometry:

```
  V3d NormalVector( Real u, Real v );
    // Returns normalized normal vector.
  StrL3d Normal( Real u, Real v ); // Returns surface normal.
  Plane TangentPlane( Real u, Real v ); // Returns tangent plane.
  virtual int GetIndicatrix( Conic3d &conic, Real u, Real v,
    Real scale = 1, int num_of_points = 120 );
    // Computes Dupin indicatrix and stores it in conic.
```

Intersection methods:

```
  void SectionWithSurface( ComplexL3d &s,
    ParamSurface &other,
    Boolean show_surfaces = false );
  // Intersect two parameterized surfaces. The intersection curve may
  // consist of several branches. Therefore, it is stored in an instance
  // s of the class ComplexL3d. If the last argument is true, a wire frame
  // model of both surfaces will be drawn in black and red. This is useful
  // for testing the solution but not for drawing or shading the surfaces.
  void GetSelfIntersection( ComplexL3d &s,
    char uv, // split either in u or v direction.
    Real t1, Real t2, Real t3, Real t4, // must be sorted: t1<t2<t3<t4
    int accuracy = 20,
    Boolean show_surfaces = false );
  // Method to compute a possible self-intersection of the surface.
  // The self-intersecion is stored in the ComplexL3d object s. It may
  // consist of several branches. In order to calculate it, you must split
  // the surface either in u-direction (uv='u') or v-direction (uv='v').
  // t1,..., t4 give the parameter limits for the two surface parts. The
  // self-intersection of these two parts will be computed. accuracy allows
  // you to set an accuracy limit, while show_surfaces = true will draw
  // the two surface parts in different colors. This is intended for a quick
  // testing of the split parameters t1,..., t4.
  void GetSelfIntersection( ComplexL3d &s,
    int n1, int n2, int n3, int n4, // must be sorted: n1 <...<n4
    Boolean show_surfaces = false );
  // Similar to preceding method. Instead of the parameter limits
  // the indices of surface facets are passed as arguments.
```

Sample code for better understanding:

```
// Compare "klein_bottle.cpp"!

#include "opengeom.h"
#include "defaults3d.h"

class KleinBottle: public ParamSurface
{
public:
    P3d SurfacePoint( Real u, Real v )
    {
        Real r = 5 *( 1 - cos( u ) / 2 );
        Real x, y, z;
        const Real a = 4, b = 15;
        if ( u < PI )
        {
            x = a * cos( u ) * ( 1 + sin( u ) ) + r * cos( u ) * cos( v );
            y = b * sin( u ) + r * sin( u ) * cos( v );
        }
        else
        {
            x = a * cos( u ) * ( 1 + sin( u ) )
                  - r * ( 1 - (u - 2 * PI ) * ( u - PI ) / 10 ) * cos( v );
            y = b * sin( u );
        }
        z = r * sin( v );
        const Real k = 0.5;
        return P3d( k * x, k * z, - k * y );
    }
};

KleinBottle Bottle;
ComplexL3d SelfIntersection;

void Scene::Init( )
{
    int n_rot_circ = 61, n_parallel_circ = 61;
    Real u1 = 0, u2 = u1 + 2 * PI;
    Real v1 = -PI, v2 = v1 + 2 * PI;
    Bottle.Def( Green, n_rot_circ, n_parallel_circ, u1, u2, v1, v2 );
    Bottle.PrepareContour( );
    ChangePScriptLineWidth( 0.5 );
}

void Scene::Draw( )
{
    Bottle.GetSelfIntersection( SelfIntersection,
```

```
    //'u', 2.5, 4, 4.2, 6.5, 10, true );
    'u', 3, 4, 4.2, 6.5, 10, true ); // only part of the curve
  return;
  SelfIntersection.Draw( Black, THICK, 1e-3 );
  // First, the bottle is shaded and equipped with
  // contour-lines and parameter lines as usual.
  Bottle.Shade( SMOOTH, SUPER_REFLECTING );
  Bottle.Contour( Black, MEDIUM );
  Bottle.WireFrame( Black, 30, 30, THIN );
  // Second, we redraw the surface transparently
  // (z-buffer deactivated)
  SetOpacity( 0.1 );
  TheCamera.Zbuffer( false );
  Bottle.Shade( SMOOTH, REFLECTING );
  Bottle.Contour( Black, THIN );
  Bottle.WireFrame( Gray, 12, 12, THIN );
  Bottle.ULines( Black, 2, MEDIUM );
  // reset defaults
  SetOpacity( 1.0 );
  TheCamera.Zbuffer( true );
}
```

Geometric transformations:

```
void Translate( Real dx, Real dy, Real dz );
void Translate( const V3d &v )
void Scale( Real kx, Real ky = 0, Real kz = 0 );
void Rotate( const StrL3d &a, Real w );
void Reflect( const Plane &p );
void Reflect( const P3d &P );
void Screw( const StrL3d &a, Real angle, Real parameter );
void Screw( const HelicalMotion &S );
```

Purely virtual functions:

```
virtual P3d SurfacePoint( Real u, Real v ); // Purely virtual!
  // When you derive a new class from ParamSurface you have to
  // implement this function when working with parametric equations.
virtual P3d SweepFunction( Real v, P3d P); // Purely virtual!
  // When you derive a new class from ParamSurface and you want
  // to define the surface as a swept surface, you have to implement
  // this virtual function.
```

class *PathCurve3d*; → declaration in `"lines3d.h"`

A curve of unknown size. Derived from *O3d* and *L3d*. Usually used for the determination of path curves during kinematic motions or animations (compare `"torus_curves1.cpp"` and `"torus_curves2.cpp"`).

Constructors and Definition:

```
PathCurve3d( Color c = Gray, int max_points = 500,
             Real critical_distance = 0 );
void Def( Color c, int max_points, Real critical_distance = 0 );
  // Allows max_points on the path curve. Path will be
  // considerd to be "closed" when the first and last points have a
  // distance less than critical_distance.
```

Additional methods:

```
void AddPoint( const P3d &P );
  // Add point P to the curve.
```

class *Plane*; → declaration in `"plane.h"`

Describes an oriented plane in 3-space. Besides various constructors, operators, and defining methods, the class is equipped with diverse methods concerning incidence relations and metric properties. Base class.

Constructors:

```
Plane( ) { } // Default constructor.
Plane( const P3d &P, const P3d &Q, const P3d &R )
  // Three noncollinear points determine a plane.
Plane( const P3d &P, const V3d &n )
  // Point plus normal vector.
Plane( const P3d &P, const StrL3d &g )
  // Point plus noncoinciding straight line.
Plane( const Rod3d &PQ ) { Def( PQ ); }
  // The plane is defined as symmetry plane of a rod.
```

Definition:

```
void Def( const P3d &P, const P3d &Q, const P3d &R );
void Def( const P3d &P, const V3d &n );
void Def( const P3d &P, const StrL3d &g );
void Def( const Rod3d &PQ );
```

Operators:

```
Plane & operator = ( const Plane &p );
  // Generate two identical planes
friend P3d operator * ( const Plane &e, const StrL3d &g );
  // Intersection of plane and straight line.
friend StrL3d operator * ( const Plane &e, const Plane &f );
  // Intersection of two planes.
```

Frequently used methods:

```
Real OrientedDistance( const P3d &P) const;
  // Return the oriented distance of a point from the plane.
Real Angle( const Plane &e ) const;
  // Return the angle between two planes (in arc length!).
```

```
Real AngleInDeg( const Plane &e ) const
   // Return the angle between two planes (degrees!).
P3d NormalProj( const P3d &P) const;
   // Return the normal projection of a point onto the plane.
void Translate( const V3d &v );
   // Translate the plane.
void Rotate( const StrL3d &a, const Real w );
   // Rotate the plane.
P3d GetPoint( ) const { return point; }
   // Return a point of the plane.
Boolean SectionWithStraightLine( const StrL3d &g,
   P3d &S, Real &t ) const;
   // Similar to the * operator, but more detailed.
Boolean SectionWithPlane( const Plane &p, StrL3d &s ) const;
   // Intersect two planes and store result in s. Return false if
   // the two planes are parallel.
Boolean SectionWithTwoOtherPlanes( const Plane &p1,
   const Plane &p2, V3d &S ) const;
   // Intersect three planes and store result in S. Return false if
   // the three planes do not have a unique intersection point.
V3d n( ) const { return normal; }
   // Return the normalized normal vector.
Real c( ) const { return constant; }
   //  Return the constant in equation px + qy + ry = c of plane.
Boolean CarrierPlane( PO3d &o, Real tol = 0 ) const;
   // Return the carrier plane of a conglomerate of 3D points.
Real z_val( Real x, Real y );
   // Project the point (x, y, 0) in z-direction onto the plane and
   // returns its z-coordinate.
```

class *Poly3d*; → declaration in `"poly3d.h"`

Closed, convex polygon in 3-space. Derived from *O3d*. Parent class for many other 3D classes (*RegPoly3d*, *Rect3d*, *Circ3d*,...). For nonconvex polygons use the class *ComplexPoly3d*.

Constructors:

```
Poly3d( ) { Texture = NULL; interiorAngle = 360; }
Poly3d( Color col, int numOfPoints, FilledOrNot filled = EMPTY )
Poly3d( Color col, int numOfPoints, Coord3dArray points,
        FilledOrNot filled = EMPTY )
```

Definition:

```
void Def( Color col, int numOfPoints, FilledOrNot filled = EMPTY );
void Def( Color col, int numOfPoints, Coord3dArray points,
        FilledOrNot filled = EMPTY );
```

Methods and operators:

```
void Draw( ThinOrThick thick, Real offset = STD_OFFSET);
```

```
    // Draw the outline.
  void Shade( );
    // Plot the filled polygon.
  void ShadeWithContour( Color col, ThinOrThick thick,
            Real offset = STD_OFFSET, Real alpha_value = 1 );
    // A combination of Draw and Shade. Additionally, the color col
    // of the outline can be specified.
  void ShadeWithTexture( TextureMap &Map,
                  Real ScaleX = 1, Real ScaleY = 1,
                  Real RotAngleInDeg = 0,
                  Real TranslateX = 0, Real TranslateY = 0,
                  Boolean repeat = true );
    // Method to map a texture on the polygon (compare
    // Section 4.4).
  void ShadeTransparent( Color interior, Real opacity,
                  Color border, ThinOrThick style );
    // Shade the polygon with transparency and draw outline;
    // colors for both interior and outline have to be specified.
  V3d GetNormalizedNormal( void ) const;
    // Return the normal vector of the supporting plane.
  Boolean SectionWithOtherPoly3d( const Poly3d &poly,
                    P3d &S1, P3d &S2, Real min_length = 1e-5 );
    // Compute the intersection line segment of two polygons.
    // Start and end points of the line segment are stored in S1
    // and S2, respectively.
```

class *Polyhedron*; → declaration in `"polyhedron.h"`

Derived from *O3d*, the class describes a 3D polyhedron in the sense of [14], Section 7.4. Its internal representation in OPEN GEOMETRY is rather complex (lists of pointers to faces, vertices, loops, plus additional information on the solid to be represented). Therefore, it is quite complicated to define a polyhedron. Actually, there doesn't even exist a simple and direct OPEN GEOMETRY defining method. However, some OPEN GEOMETRY classes provide methods such as TransformIntoPolyhedron( ) that automatically generate polyhedra.

Constructors:

```
Polyhedron( ); // Default constructor.
```

Methods:

```
void Def( Polyhedron &p ); // Define via other polyhedron p.

// Methods for drawing and shading
void WireFrame( int modus = 0 /* all edges */,
      Boolean RecalcEdges = false,
      ThinOrThick linestyle = THIN,
      Real offset = STD_OFFSET );
```

```
void Shade( Color col, Boolean reflect = true ,
            FaceMode modus = ALL_FACES );
void ShadeSmoothButKeepEdges( Color col, Boolean reflect = true ,
            FaceMode modus = ALL_FACES );
   // Shade smoothly only if facets intersect in a reasonably small
   // angle 0.7 < Global.SmoothLimit < 0.99. The user can directly
   // access and change this global variable (compare Figure 6.17).
void ShadeWithTexture( Color col, Boolean reflect, FaceMode modus,
            TextureMap &map );
void Contour( Color f, ThinOrThick thick, Real offset = 1e-3 );

// Geometric transformations (overwrite methods of O3d):
void Rotate( const StrL3d &a, Real w );
void Rotate( const RotMatrix &matrix );
void Reflect( const Plane &p );
void Reflect( const P3d &c );
void Scale( Real kx, Real ky, Real kz );
void Scale ( Real k );
void Translate( const V3d &t );
void Translate( Real x, Real y, Real z );

// Further useful methods:
int nFaces( ); // Return the number of faces.
void PlotShadow( /* const */ Poly3d &plane, Real offset = 1e-2 );
void PlotShadowOnHorizontalPlane( Color col, Real z );
void GetMinMaxZ( Real &zmin, Real &zmax );
   // Calculate minimal and maximal z-coordinate of all vertices.

void SectionWithPlane( Plane &p, ComplexL3d &s );
void SectionWithPolyhedron( ComplexL3d &sextion, Polyhedron &T,
            Boolean show = false );
   // Calculate the intersection of two polyhedra and store them
   // in s. If show is true, wire frame models of both polyhedra will
   // be displayed.
void GetSelfIntersection( ComplexL3d &s,
   int n1, int n2, int n3, int n4, // must be sorted: n1 <n2<n3<n4
   Boolean show_surfaces = false );
// Method to compute a possible self-intersection of the polyhedron.
// The self-intersecion is stored in the ComplexL3d object s. It may
// consist of several branches. You have to specify the indices of the
// relevant facets of the polyhedron. If show_surfaces = true, the
// corresponding parts of the polyhedron will be drawn in different
// colors. This is intended for a quick testing of the split
// indices n1,..., n4.
```

class *Projectivity*; → declaration in `"proj_geom.h"`

This class consists of two classes of type *ProjScale* and provides methods to deal with the projectivity between them. Note that some of the

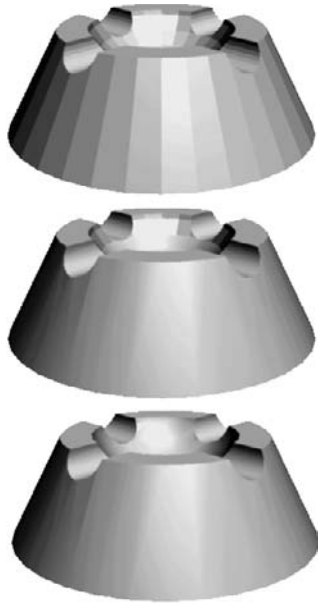

FIGURE 6.17. An ashtray (polyhedron converted from a CAD3D object) is shaded with different smooth limits s. In case of $s = 0.99$ (top), all edges are visible. If $s = 0.98$ (middle), you can see some edges, while, in case of $s = 0.94$ (bottom), everything is completely smooth except the intended sharp edges.

methods make sense only if the projective scales are of the right type. Base class.

Constructors:

Projectivity( ); // Default constructor.

Definition:

void Def(*ProjScale* &InScale1, *ProjScale* &InScale2,
const *Real* eps = 1e−6, **const** *Real* infinity = 1e6);
// Cross ratios of absolute value eps or less will
// be treated as zero. Cross ratios of absolute value
// infinity will be set to infinity.

Methods and operators:

P3d MapPointToPoint(**const** *P3d* &A);
StrL3d MapPointToLine(**const** *P3d* &A);
Plane MapPointToPlane(**const** *P3d* &A);
P3d MapLineToPoint(**const** *StrL3d* &a);
StrL3d MapLineToLine(**const** *StrL3d* &a);
Plane MapLineToPlane(**const** *StrL3d* &a);

```
P3d MapPlaneToPoint( const Plane &p );
StrL3d MapPlaneToLine( const Plane &p );
Plane MapPlaneToPlane( const Plane &p );
   // Nine methods to map elements of the first scale
   // to elements of the second scale. The method to be
   // used depends on the type of the projective scales.
P3d InvMapPointToPoint( const P3d &A );
StrL3d InvMapPointToLine( const P3d &A );
Plane InvMapPointToPlane( const P3d &A );
P3d InvMapLineToPoint( const StrL3d &a );
StrL3d InvMapLineToLine( const StrL3d &a );
Plane InvMapLineToPlane( const StrL3d &a );
P3d InvMapPlaneToPoint( const Plane &p );
StrL3d InvMapPlaneToLine( const Plane &p );
Plane InvMapPlaneToPlane( const Plane &p );
   // Nine methods to map elements of the second scale
   // to elements of the first scale. The method to be
   // used depends on the type of the projective scales.
void MarkElements( Color col, const Real rad1,
      const Real rad2 = 0 );
void MarkElements( Color col[3], const Real rad1,
      const Real rad2 = 0 );
   // Two methods to mark the base points on either of the
   // two projective scales, either in one color or in three
   // different colors according to the respective kind of point
   // (origin, unit "point", "point" at infinity).
void ConnectElements( Color col, ThinOrThick thick );
void ConnectElements( Color col[3], ThinOrThick thick );
   // Two methods to connect corresponding base points on the
   // two projective scales. The connecting lines will be either
   // drawn in one color or in three different colors according
   // to the respective kind of point (origin, unit "point",
   // "point" at infinity).
StrL3d IntersectPlanes( Real ratio );
   // Method to intersect corresponding planes.
StrL3d ConnectingLine( Real ratio );
   // Method to connect corresponding points.
ProjType GetType1( ) { return Type1; }
ProjType GetType2( ) { return Type2; }
   // Two methods to return the respective types of the projective
   // scales. Possible return values are PencilOfLines,
   // PencilOfPoints, PencilOfPlanes, PointsOnConic,
   // TangentsOfConic, and TangentPlanesOfCone.
```

class *ProjScale*; → declaration in `"proj_geom.h"`

Describes a projective scale on either a pencil of points, a pencil of lines, a pencil of planes, the point set of a conic, the tangent set of a conic, or

the set of tangent planes of a cone of second order. The scale consists of a supporting element (straight line, point, or conic) and a projective coordinate system (origin, "point" at infinity and unit "point"[2]). Base class.

Constructors:

```
ProjScale( ); // Default constructor.
```

Definition:

```
ProjScale & operator = ( ProjScale &other );
void Def( const P3d &V, StrL3d t[3] ); // pencil of lines
void Def( const StrL3d &l, P3d Q[3] ); // pencil of points
void Def( const StrL3d &l, Plane p[3] ); // pencil of planes
void Def( Conic3d &c, P3d Q[3] ); // points on conic
void Def( Conic3d &c, StrL3d t[3] ); // tangents of conic
void Def( const P3d &V, Conic3d &c, Plane p[3] );
   // tangent planes of cone of second order
```

Methods and operators:

```
void Draw( Color col, Real u0, Real u1, ThinOrThick thick );
   // Drawing method for pencil of points.
void Draw( ThinOrThick thick );
   // Drawing method for points or tangents of conic and
   // for tangent planes of cone of second order.
void Mark( Color col, Real rad1, Real rad2 = 0 );
   // Drawing method for pencil of lines.
void MarkElement( Color col, Real rad1, Real rad2, int i );
void MarkElements( Color col, Real rad1, Real rad2 = 0 );
   // Two methods for marking origin (i = 0), unit point (i = 1),
   // and point at infinity (i = 2) if these elements are points.
void DrawElement( Color col, Real u0, Real u1,
          ThinOrThick thick, int i );
void DrawElements( Color col, Real u0, Real u1, ThinOrThick thick );
   // Two methods for drawing origin (i = 0), unit point (i = 1),
   // and point at infinity (i = 2) if these elements are straight lines.
ProjType GetType( ) { return Type; }
   // Returns the type of the projective scale. Possible return values
   // are PencilOfLines, PencilOfPoints, PencilOfPlanes,
   // PointsOnConic, TangentsOfConic, and TangentPlanesOfCone.
void SetElement( const StrL3d &t, int i ) { s[i] = t;
     ComputeReals( ); }
void SetElement( const P3d &Q, int i ) { P[i] = Q;
     ComputeReals( ); }
void SetElement( const Plane p, int i ) { plane[i] = p;
     ComputeReals( ); }
```

[2]In this context, the word "point" means either really a point, a straight line, or a plane.

```
    // Three methods for redefining the projective scale by setting
    // the origin, unit "point" or "point" at infinity.
    // ComputeReals( ) is a private method of the class that
    // computes some necessary data.
P3d GetPoint( int i ) { return P[i]; }
StrL3d GetLine( int i ) { return s[i]; }
Plane GetPlane( int i ) { return plane[i]; }
    // Three methods for returning origin, unit "point"
    // or "point" at infinity.
P3d PointOfCrossRatio( const Real ratio );
StrL3d LineOfCrossRatio( const Real ratio );
Plane PlaneOfCrossRatio( const Real ratio );
    // Returns the point, straight line, or plane, respectively,
    // that determines the given cross ratio with respect to the
    // projective scale.
Real CrossRatio( const P3d &A, const Real eps = 1e-6,
      const Real infinity = 1e6 );
Real CrossRatio( const StrL3d &a, const Real eps = 1e-6,
      const Real infinity = 1e6 );
Real CrossRatio( const Plane &p, const Real eps = 1e-6,
      const Real infintiy = 1e6 );
    // Returns the cross ratio that a point, straight line, or plane,
    // respectively, determines on the projective scale. If its
    // absolute value is smaller than eps or larger than infinity,
    // the return value will be 0 or infinity itself.
```

Sample code for better understanding:

```
ProjScale Scale;
StrL3d axis;
axis.Def( Origin, Zdir );
Plane plane[3];
plane[0].Def( P3d( 5, 0, 0 ), axis );
plane[1].Def( P3d( 0, 5, 0 ), axis );
plane[2].Def( P3d( 5, 5, 0 ), axis );
Scale.Def( axis, plane );
Scale.Draw( Black, -15, 15, VERY_THICK );
Plane pi;
Real ratio = -1;
pi = Scale.PlaneOfCrossRatio( ratio );
if ( !( Scale.CrossRatio( pi ) == -1 ) )
   Write( "numerical problem" );
```

class *Quadric*; → declaration in `"ruled_surface.h"`

Describes a ruled surface of order two (quadric) in 3-space. The quadric is defined by three pairwise skew straight lines r[0], r[1], and r[2] (generators of second kind). The generators of first kind are those lines that intersect r[0], r[1], and r[2]. The class is derived from *RuledSurface* and *ParamSurface*.

Definition:

```
void Def( Color c, int m, int n, Real u1, Real u2,
          Real v1, Real v2, StrL3d r[3] );
// u1 and u2 are the parameter limits on r[0];
// v1 = 0 and v2 = 1 yield the part between r[0] and r[2].
```

Virtual member functions:

```
virtual P3d DirectrixPoint( Real u );
virtual V3d DirectionVector( Real u );
// DirectrixPoint( u ) equals r[0].InBetweenPoint( u )
```

Frequently used methods and operators:

```
StrL3d Ruling1( Real u ); // Rulings of first kind.
StrL3d Ruling2( Real v ); // Rulings of second kind.
void DrawRulings1( Color c, int n, ThinOrThick thick,
     Real offset = STD_OFFSET );
void DrawRulings2( Color c, int n, ThinOrThick thick,
     Real offset = STD_OFFSET );
void DrawRulings( Color c, int m, int n, ThinOrThick thick,
     Real offset = STD_OFFSET );
```

class *RatBezierCurve3d*; → declaration in `"bezier.h"`

Describes a rational Bézier curve in 3D by its control polygon and its weights. The curve points are computed by means of DeCasteljau's algorithm. We provide methods for manipulating the curve (changing of control points and weights, degree elevation, splitting). The base class is *ParamCurve3d*. Some methods are overwritten.

Constructors:

```
RatBezierCurve3d( ) { B = NULL; C = NULL; w = NULL; }
// B and C are dynamically allocated arrays of control points,
// w is a dynamically allocated array of reals.
```

Definition:

```
void Def( Color c, int total_size, int size_of_control_poly,
          P3d Control[], Real weigth[] );
// total_size is the number of curve points that will
// be computed, while size_of_control_poly is the number
// of control points to be used.
```

Frequently used methods and operators:

```
virtual P3d CurvePoint( Real u )
{
   return DeCasteljau( u );
}
virtual StrL3d Tangent( Real u )/* const = 0 */;
virtual Plane OsculatingPlane( Real u )/* const = 0 */;
void DrawControlPolygon( Color col, ThinOrThick thickness );
void MarkControlPoints( Color col, Real rad1, Real rad0 );
void SetControlPoint( int i, P3d P );
   // Redefine control point with index i.
void SetWeight( int i, Real w );
   // Redefine weight with index i.
P3d GetControlPoint( int i );
   // Return control point with index i.
Real GetWeight( int i );
   // Return weight i.
int GetPNum( ) { return PNum; }
   // PNum is the number of control points.
void ElevateDegree( int i ); // Elevates the degree by i.
void ReversePolygon( void );
   // Reverse the order of the control points
   // (useful after splitting of the curve).
void Split( Real u, Boolean second_half );
   // Split the Bézier curve at the parameter value u.
   // Depending on the value of second_half, the new
   // curve is identical to the part of the old curve that
   // corresponds to the parameter interval [0, u] or [u, 1].
```

Sample code for better understanding:

```
RatBezierCurve3d BezierCurve;
P3d P[3];
P[0]( 0, 0, 0 ), P[1]( 4, 0, 4 ), P[2]( 0, 4, 8 );
Real weight[3];
weight[0] = 2, weight[1] = 1, weight[2] = 3;
BezierCurve.Def( Black, 200, 3, BasePoints, weight );
BezierCurve.ElevateDegree( 2 );
BezierCurve.Split( 0.5, true );
BezierCurve.Draw( VERY_THICK );
BezierCurve.DrawControlPolygon( Blue, THICK );
BezierCurve.MarkControlPoints( Blue, 0.15, 0.1 );
```

class *RatBezierSurface*; → declaration in `"bezier.h"`

Describes a rational Bézier surface by its control polygon and its weights. The curve points are computed by means of DeCasteljau's algorithm. We provide methods for manipulating the surface (changing of control points and weights, degree elevation, splitting). The base class is *ParamSurface*.

Constructors:

RatBezierSurface( ) { B = BB = NULL; w = ww = NULL; }
// B and BB are dynamically allocated 2D arrays of control points,
// w and ww are dynamically allocated arrays of reals.

Definition:

void Def(*Color* c, **int** m, **int** n, **int** pnum1, **int** pnum2,
P3d **Control, *Real* **weight);
// m and n determine the number of facets in u- and v-directions.
// pnum1 and pnum2 are the dimensions of the control net.

Frequently used methods and operators:

P3d SurfacePoint(*Real* u, *Real* v) { **return** DeCasteljau(u, v); }
void DrawControlPolygon(*Color* col, *ThinOrThick* style,
Real offset = *STD_OFFSET*);
void MarkControlPoints(*Color* col, *Real* rad1, *Real* rad0 = 0);
P3d GetControlPoint(**int** i, **int** j) { **return** B[i][j]; };
Real GetWeight(**int** i, **int** j) { **return** w[i][j]; };
void SetControlPoint(**int** i, **int** j, *P3d* P);
// Redefine control point with indices i and j.
void SetWeight(**int** i, **int** j, *Real* weight);
// Redefine weight with indices i and j
int GetPNum1( ) { **return** PNum1; };
int GetPNum2( ) { **return** PNum2; };
// Two methods that return the dimension of the control net.
void ElevateDegree(**int** m, **int** n);
void ReversePolygon(*Boolean* u_dir, *Boolean* v_dir);
// Reverse the order of the control net in u- and/or
// v-direction according to the value of the Boolean arguments.
void SplitU(*Real* u, *Boolean* second_half);
// Split the Bézier surface at the parameter value u.
// Depending on the value of second_half, the new
// curve is identical to the part of the old curve that
// corresponds to the parameter interval $[0, u]$ or $[u, 1]$.
void SplitV(*Real* v, *Boolean* second_half);
// Analogous to SplitU(...).

Sample code for better understanding:

```
RatBezierSurface BezSurf;
P3d **P = NULL;
Real **weight = NULL;
ALLOC_2D_ARRAY( P3d, P, 3, 3, "P" );
ALLOC_2D_ARRAY( Real, weight, 3, 3, "weight" );
P[0][0].Def( -8, -8, -4 );
P[0][1].Def( -8,  0, 4 );
P[0][2].Def( -8,  8, -4 );

P[1][0].Def( -2, -8, 5 );
P[1][1].Def( -2,  0, 8 );
P[1][2].Def( -2,  8, 5 );

P[2][0].Def(  2, -8, 5 );
P[2][1].Def(  2,  0, 8 );
P[2][2].Def(  2,  8, 5 );

int i, j;
for ( i = 0; i < m; i++ )
    for ( j = 0; j < n; j++ )
        weight[i][j] = 1;

BezSurf.Def( Blue, 31, 61, 4, 3, P, weight );

BezSurf.SplitU( 0.5, true );
BezSurf.SplitV( 0.5, false );
BezSurf.ElevateDegree( 1, 2 );
BezSurf.SetControlPoint( 0, 0, P3d( -12, -8, 3 ) );
BezSurf.SetWeight( 0, 0, 20 );

FREE_2D_ARRAY( P3d, P, m, n, "P" );
FREE_2D_ARRAY( Real, weight, m, n, "weight" );

BezSurf.Shade( SMOOTH, REFLECTING );
```

class *Rect3d*; → declaration in `"poly3d.h"`

Rectangle in 3-space. Inherits methods of *Poly3d*. In the starting position two rectangle sides lie on the x- and y-axes of the coordinate system, and one vertex coincides with the origin.

Constructors and definition:

```
Rect3d( );
Rect3d( Color col, Real length, Real width,
    FilledOrNot filled = EMPTY )
void Def( Color col, Real length, Real width,
    FilledOrNot filled = EMPTY );
```

class *RegDodecahedron*; → declaration in `"dodecahedron.h"`

Describes a regular dodecahedron (polyhedron consisting of twelve regular pentagons) and provides methods for drawing and shading. Derived from *O3d*. In order to use this class, you must include `"dodecahedron.h"` in your OPEN GEOMETRY file.

Definition:

```
void Def( Color c, Real side );
```

Frequently used methods and operators:

```
Real GetSide( ) { return a; }
Real GetRadius( );
Real GetThickness( );
P3d GetCenter( );
void WireFrame( Color c, ThinOrThick style );
void Shade( );
void GetCircumSphere( Sphere &s );
void GetEdge( int i, P3d &a, P3d &b );
void GetNeighbors( int i, P3d n[3] );
void GetFace( int i, Poly3d &poly );
```

Sample code for better understanding:

```
#include "dodecahedron.h"

RegDodecahedron D;
void Scene::Init( )
{
   D.Def( Blue, 3 );
   D.Translate( -D.GetCenter( ) );
   D.Rotate( Zaxis, 15 );
}
void Scene::Draw( )
{
```

```
    D.WireFrame( Gray, (ThinOrThick) 3 );
    Poly3d p;
    for ( int i = 1; i <= 6; i++ ) // shade only bottom faces
    {
       D.GetFace( i, p );
       p.Shade( );
    }
}
```

FIGURE 6.18. A regular dodecahedron (compare sample code).

class *RegFrustum*; → declaration in `"frustum.h"`

Describes a frustum in 3-space. Inherits the methods and operators of *O23d* and *O3d*. Base class for *RegPrism* and *RegPyramid*. A closer description can be found in [14], pp. 132 ff.

Constructors:

```
RegFrustum( ); // Default constructor.
RegFrustum( Color col, Real r1, Real r2, Real height,
        int n, SolidOrNot isSolid = SOLID,
        const StrL3d &axis = Zaxis );
   // h = height of frustum,
   // r1, r2 = radii of base and top circle, respectively,
   // n = number of points on polygon that approximates base
   // circle; low value of n results in frustums of pyramid.
```

Definition:

```
void Def( Color col, Real r1, Real r2, Real height,
      int n, SolidOrNot = SOLID, const StrL3d &axis = Zaxis );
```

```
    // See constructor.
```

Additional or overwriting methods and operators:

```
// Some shading and drawing methods:
void Shade( FlatOrSmooth smooth = SMOOTH,
                    Boolean reflect = true );
void ShadeBackfaces( FlatOrSmooth smooth = SMOOTH,
                    Boolean reflect = true );
void WireFrame( Boolean remove_hidden_lines, ThinOrThick thick,
                    Real offset  = STD_OFFSET ); /* const */
void Outline( Color col, ThinOrThick style,
                    Real offset = STD_OFFSET );
void DrawBottom( Color col, ThinOrThick style,
                    Real offset = STD_OFFSET );
void DrawTop( Color col, ThinOrThick style,
                    Real offset = STD_OFFSET );

// "Getters" and "setters":
int GetOrder( ) const; // Return the "order" n as specified in Def).
StrL3d GetGeneratingLine( int i ); // Return the i-th generating line.
StrL3d GetAxis( ) const; // Return frustum axis.
StrL3d GetAxis( P3d &M1, P3d &M2 ) const;
    // Return frustum axis and store centers of
    // top and base in M1 and M2, respectively.
void GetRadii( Real &r1, Real &r2 ) const;
    // Store radii of top and base in r1 and r2, respectively.
void SetColorOfBaseAndTop( Color c );
    // Allow to shade base and top in different color.

// Transformation methods:
void Rotate( const StrL3d &axis, Real angle_in_deg );
void Scale( Real kx, Real ky, Real kz );
void TranslateTop( const V3d &t );
void RotateTop( const StrL3d &axis, Real angle_in_deg );
    // If you destroy the convexity of the object by the above
    // transformations, use one of the following before shading it!
void Transform( Polyhedron &P );
void TransformIntoPolyhedron( );

// Boolean operations for frustums:
friend Solid * operator + ( RegFrustum &One,
                                    RegFrustum &Two );
friend Solid * operator - ( RegFrustum &One,
                                    RegFrustum &Two );
```

class *RegPoly3d*; → declaration in `"poly3d.h"`

Closed regular polygon in 3-space. Derived from *Poly3d*.

Definition:

```
void Def( Color col, const P3d &FirstElem, const StrL3d &axis,
      int n, FilledOrNot = EMPTY );
   // A vertex and the axis are given.
void Def( Color col, const P3d &mid, const V3d &ax_dir, Real r,
      int n, FilledOrNot = EMPTY );
   // Center, axis direction, and radius are given.
   // First vertex lies on a horizontal line through the center.
```

Public member variables:

```
StrL3d axis;
Real rad;
P3d Middle;
```

Methods:

```
int SectionWithPlane( Plane &e, P3d &S1, P3d &S2 );
   // Compute the intersection line segment with a plane. Start
   // and end points of the line segment are stored in S1 and S2,
   // respectively.
P3d GetCenter( ) const // Return the center.
Real GetRadius( ) const // Return the radius.
StrL3d GetAxis( ) const // Return the axis.
void Rotate( const StrL3d &a, Real w );
   // Rotate polygon about axis a through w°.
```

class *RegPrism*; → declaration in `"frustum.h"`

Describes a regular prism in 3-space. Derived from *RegFrustum*, it differs from this class only with respect to constructors and defining methods.

Constructors and Definition:

```
RegPrism( );
RegPrism( Color col, Real radius, Real height,
          int n, SolidOrNot isSolid = SOLID,
          const StrL3d &axis = Zaxis );
void Def( Color col, Real radius, Real height,
   int n, SolidOrNot = SOLID, const StrL3d &axis = Zaxis );
   // n is the number of points on base and top polygon.
```

class *RegPyramid*; → declaration in `"frustum.h"`

Describes a regular pyramid in 3-space. Derived from *RegFrustum*, it differs from this class only with respect to constructors and defining methods.

Constructors and Definition:

```
RegPyramid( ); // Default constructor.
RegPyramid( Color col, Real radius, Real height,
```

```
                int n, SolidOrNot isSolid = SOLID,
                const StrL3d &axis = Zaxis )
    void Def( Color col, Real radius, Real height, int n,
          SolidOrNot = SOLID, const StrL3d &axis = Zaxis );
       // n is the number of points on base polygon.
    P3d Apex( ); // Return the apex.
```

class *Rod3d*; → declaration in `"poly3d.h"`

Describes a rod in 3-space. A rod may be imagined as a polygon, two vertices or a line segment. It is used mainly for kinematic animations and, sometimes, for the definition of the plane of symmetry of two points.

Constructors and Definition:

```
Rod3d( Color f, const P3d &A, const P3d &B )
void Def( Color f, Real x1, Real y1, Real z1,
   Real x2, Real y2, Real z2 );
void Def( Color f, const P3d &P, const P3d &Q );
```

Additional methods (see also Poly3d):

```
void Dotted( int n );
void LineDotted( int n );
void Draw( ThinOrThick thick, Real offset = STD_OFFSET );
```

class *RuledSurface*; → declaration in `"ruled_surface.h"`

Describes a ruled surface by means of its directrix curve and direction vectors. It is an *abstract class* derived from *ParamSurface*. In order to define a specific surface, the user has to derive an additional class from this class! There, the virtual functions DirectrixPoint(...) and DirectionVector(...) have to be implemented.

Constructors:

```
RuledSurface( ) { } // Default constructor.
```

Virtual member functions:

```
virtual P3d DirectrixPoint( Real u ) = NULL;
   // Has to be implemented by the user.
virtual V3d DirectionVector( Real u ) = NULL;
   // Has to be implemented by the user.
virtual P3d SurfacePoint( Real u, Real v );
   // Is already implemented.
```

Frequently used methods and operators:

```
void GetDirectrix( L3d &directrix, Color col, int size );
   // Assign the directrix curve to an L3d object.
void DrawDirectrix( Color col, int n, ThinOrThick thick,
```

```
        Real offset = 1e−3 ); // Draws the directrix curve.
    StrL3d Generator( Real u );
      // Return the generating line at parameter u.
    P3d PedalPoint( Real u, P3d Pole );
      // Return the normal projection of Pole onto
      // the generator at parameter u (pedal point).
    void GetPedalLine( L3d &pedal_line, P3d Pole, Color col, int size );
      // Assign the pedal line of Pole to an L3d object.
    void DrawPedalLine( P3d Pole, Color col, int n,
        ThinOrThick thick, Real offset = 1e−3 );
      // Draw the pedal line of Pole.
    P3d CentralPoint( Real u );
      // Return the central point (point of striction)
      // on the generator at parameter u.
    void GetCentralLine( L3d &line, Color col, int size );
      // Assign the central line to an L3d object.
    void DrawCentralLine( Color col, int n, ThinOrThick thick,
        Real offset = 1e−3 ); // Draws the central line.
    V3d OscQuadrDir( Real u, Real v, Real eps );
      // Return the direction of the generator of the osculating quadric
      // in the surface point at parameters u and v. It is computed as the
      // intersection line of two neighboring generators corresponding to
      // u ± eps and can be used to define the osculating quadric as an
      // instance of the class Quadric.
```

class *Sector3d*; → declaration in `"arc3d.h"`

Class for drawing a sector of a 3D circle. It provides various defining methods. The sector is either filled or not and the user can decide whether to draw the line segments that connect the center with start and end points (value of only_segment = **false** or **true**; compare Figure 6.19). *Sector3d* is derived from *Circ3d*.

Constructors:

```
Sector3d( ) { }; // Default constructor.
```

Definition:

```
void Def3Points( Color col, const P3d &P, const P3d &Q,
    const P3d &R, int n, FilledOrNot filled = EMPTY );
    // Define sector via three points. The method name differs from
    // standard Open Geometry conventions in order to be consistent
    // with the class Sector2d.
void Def( Color col, const P3d &FirstElem, const StrL3d &a, Real w,
    int n, FilledOrNot = EMPTY );
    // Define sector via start point, axis, and central angle. Note that
    // the apex angle will always be less than 180°!
void Def( Color col, P3d &FirstElem, StrL3d &a, P3d &LastElem, int n,
    FilledOrNot = EMPTY );
    // Define sector via start point, axis, and end point.
```

Frequently used methods and operators:

```
void Draw( Boolean only_segment /*= false*/,
    ThinOrThick thick /* = false */, Real offset = STD_OFFSET );
// If only_segment is false, the connecting lines of
// center with start and end points will be displayed.
void Shade( );
// Shades the sector; has the same effect as Draw(...) if the
// sector has been defined as FILLED.
void ShadeWithContour( Color c, ThinOrThick thick,
    Real alpha_value = 1 );
// The same as Shade( ); additionally, the contour is displayed.

// The meaning of the remaining methods should be obvious:
Real GetCentralAngleInDeg( );
Real GetRadius( );
P3d GetCenter( );
P3d GetFirstPoint( );
P3d GetLastPoint( );
P3d GetPointOnBisectrix( );
void ChangeRadius( Real new_radius );
```

For an example, have a look at the following sample code excerpt from `"two_flight_routes.cpp"`. It displays the routes of two airplanes on Earth and is a nice demonstration for the use of *Sector3d* (compare also Figure 6.19).

Sample code for better understanding:

```
#include "opengeom.h"

const Real Rad = 6.37, Height = 0.5 * Rad;
Circ3d R1, R2;
P3d A, B, C, D;
Sphere S;
Sector3d AB, CD;
StrL3d Intersection;

void Scene::Init( )
{
  P3d M = Origin;
  S.Def( Blue, M, Rad, 50 );
  A( -20, 20, 0 );
  B( 90, 20, 70 );
  C( 70, 70, 20 );
  D( 10, -10, 60 );
  Real t = Rad + Height;
  A *= t / A.Length( );
```

```
    B *= t / B.Length( );
    C *= t / C.Length( );
    D *= t / D.Length( );
    t = Rad / t;
    P3d A1 = -1.0 * A, C1 = -1.0 * C;
    R1.Def( Green, A, B, A1, 100 );
    R2.Def( Red, C, D, C1, 100 );
    P3d N = 0.5*(A+B);
    N *= ( Rad + Height ) / N.Length( );
    AB.Def3Points( Black, A, N, B, 40, EMPTY );
    N = 0.5*(C+D);
    N *= ( Rad + Height ) / N.Length( );
    CD.Def3Points( Black, C, N, D, 40, EMPTY );
    Plane e1( A, B, M ), e2( C, D, M );
    e1.SectionWithPlane( e2, Intersection );
}
void Scene::Draw( )
{
    Circ3d c;
    S.GetContour( c );
    c.Draw( MEDIUM );
    c.ChangeColor( PureWhite );
    c.Shade( );
    R1.Draw( THIN, 0 );
    R2.Draw( THIN, 0 );
    AB.Draw( true, THICK );
    CD.Draw( true, THICK );
    showCircle( Black, A, B );
    showCircle( Black, C, D);
    showPoint( Green, A );
    showPoint( Green, B );
    showPoint( Red, C );
    showPoint( Red, D );
}
```

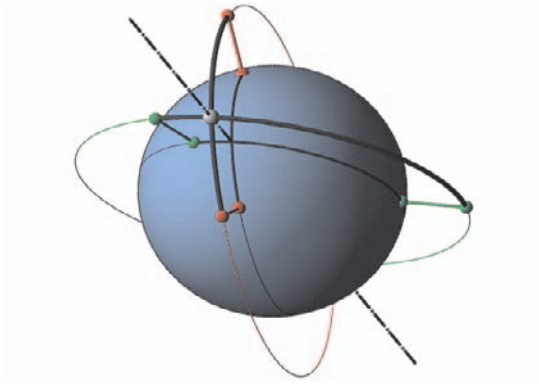

FIGURE 6.19. Different flight routes on Earth. The corresponding program `"two_flight_routes.cpp"` makes extensive use of the class *Sector3d*.

class *Solid*; → declaration in `"solid.h"`

Describes a solid (and therefore always closed) polyhedron in CAD3D format. Can be read from data file or generated by OPEN GEOMETRY with the help of Boolean operators. Note that the result of a Boolean operation is a pointer to a *Solid* called *BoolResult*. In the sample file you can see how to work with such pointers.

Constructors:

```
Solid( ); // Default constructor.
// Destricption of the below constructors: see Def(...).
Solid( RegFrustum &frustum );
Solid( Box &box );
Solid( const char *name ); // Name of a CAD3D−file.
Solid( ParamSurface &ClosedSurf, Boolean IsSurfOfRevol );
Solid( FunctionGraph &Graph ); // Solid under a function graph.
```

Definition:

```
// The standard way to get a Solid = CAD3D object
void ReadFrom_CAD3D_File( const char *name );
// You can also initialize Solids with given
// OPEN GEOMETRY objects:
void Def( RegFrustum &frustum );
void Def( Box &box );
void Def( Polyhedron &phdr );
void Def( ParamSurface &ClosedSurf, Boolean IsSurfOfRevol );
void Def( FunctionGraph &Graph );
void Def( const char *name );
```

Operators:

```
// Boolean operators for "addition", "difference", "intersection":
```

```
friend BoolResult operator + ( const Solid &One,
                               const Solid &Two );
friend BoolResult operator − ( const Solid &One,
                               const Solid &Two );
friend BoolResult operator ∗ ( const Solid &One,
                               const Solid &Two );
friend BoolResult operator ∗ ( const Solid &Total,
                               const Plane &plane );
```

Methods:

```
void Shade( FlatOrSmooth smooth = SMOOTH,
      Boolean reflect = REFLECTING,
      FaceMode fmode = ONLY_FRONTFACES );
void WireFrame( Color c, Boolean remove_hidden_lines,
      ThinOrThick thick, Real offset = STD_OFFSET );
   // Removal of invisible edges are done by first plotting
   // the object in background color
int NoOfPolys( );
   // Return number of faces.
void SetColor( Color c, int n1 = 0, int n2 = 0 );
  // Change the color of the faces from index n1 to index n2.
   // Non-default n1 starts with 1, max. n2 is NoOfPolys.
   // The default values change the color of the entire solid
   // (same as ChangeColor( )).
void ChangeColor( Color c );

// Geometric transformations:
void Translate( Real x, Real y, Real z );
void Translate( const V3d &t );
void Rotate( const StrL3d &a, Real angle_in_deg );
void Rotate( const RotMatrix &R );
void Scale( Real sx, Real sy, Real sz );
void Scale( Real s );
void Reflect ( const Plane &p );

// Export:
void SaveAs_CAD3D_File( const char ∗name );
   // Create a CAD3D file
void ExportTo_POV_Ray( const char ∗name );
   // Export to POV-Ray for high-quality output
void Delete( Boolean file = false );
   // file = true will delete the data file!
```

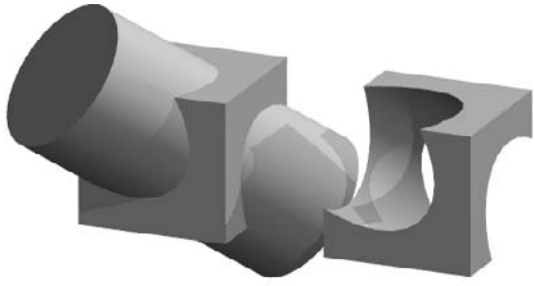

FIGURE 6.20. Output of the sample file for *Solid*.

Sample code for better understanding:

```
#include "opengeom.h"
#include "defaults3d.h"

BoolResult Union, Difference;

void Scene::Init( )
{
   // Define a first solid by means of a cube
   Box cube;
   cube.Def( Gray, P3d( −3, −3, −3 ), P3d( 3, 3, 3 ) );
   Solid FirstSolid( cube );
   // Define a second solid by means of a cylinder
   RegPrism cyl;
   StrL3d axis( Origin, V3d( 1, −1, 1 ) );
   cyl.Def( Gray, 3.2, 12, 35, SOLID, axis );
   cyl.Translate( −6 ∗ axis.GetDir( ) );
   Solid SecondSolid( cyl );
   // Now define union and differences
   Union = SecondSolid + FirstSolid;
   Union−>Translate( 0, −5, 0 );
   Difference = FirstSolid − SecondSolid;
   Difference−>Translate( 0, 5, 0 );
   ChangeSmoothLimit( 0.98 );
   TheCamera.SetDepthRange(−15, 15);
}

void Scene::Draw( )
{
   Union−>Shade( );
   Difference−>Shade( );
}
```

class *Sphere*; → declaration in `"sphere.h"`

Describes a sphere in 3-space. The class inherits methods from *Circ3d* and *O3d*.

Constructors:

```
Sphere( ); // Default constructor.
Sphere( Color col, P3d &M, Real rad, int n = 100,
        int points_on_meridian = 0, int order = 0 );
  // If the number of points on the meridian and the order are not
  // specified, the program tries to optimize these numbers.
  // In general, one should increase them for larger spheres. The
  // same is true for the number of points on the contour (which
  // is usually a small circle).
```

Definitions:

```
void Def( Color col, const P3d &M, Real rad, int n = 100,
          int points_on_meridian = 0, int order = 0 );
  // See constructor.
```

Additional or overwriting methods and operators:

```
void Draw( Color col, ThinOrThick thick,
           Real offset = STD_OFFSET );
  // Draw the silhouette, i.e., the image of the contour.
void Shade( Shininess reflect = REFLECTING );
  // Ordinary shading.
void WireFrame( Color col, ThinOrThick style,
      int n1 = 0, int n2 = 0,
      Real offset = STD_OFFSET );
  // Draw a wire frame with n1 meridians
  // and n2 parallel circles.
void Scale( Real k );
  // Only one parameter. Other scaling methods are not
  // allowed. Ellipsoids can be plotted as parameterized surfaces.

// some congruence transformations
void Translate( const V3d &t );
void Translate( Real x, Real y, Real z );
void Rotate( const StrL3d &axis, Real deg );

P3d GetCenter( ) const{ return Middle; } ;
Real GetRadius( ) const { return radius; };

Plane PolarPlane( P3d &P) const;
  // Return the polar plane of a point P.
Boolean SectionWithPlane( const Plane &e, Circ3d &c,
      int size_of_c = 100 ) const;
  // Return true if there is a section. The intersection circle c
  // is calculated with a default number of 100 points.
```

```
Boolean SectionWithSphere( const Sphere &s, Circ3d &c,
      int size_of_c = 100 ) const;
   // Return true if there is a section. The intersection circle c
   // is calculated with a default number of 100 points.
int SectionWithStraightLine( const StrL3d &s, P3d &A,
                                                P3d &B ) const;
   // Return the number of intersection points with the straight
   // line s. The intersection points are stored in A and B.
void GetContour( Circ3d &c );
   // Compute the contour for the current projection (a circle)
   // and store it in c.
```

class *StrL3d*; → declaration in `"strl3d.h"`

StrL3d describes an oriented straight line in 3-space. It has already been described in [14], p. 84. However, for your convenience, we list all methods and operators.

Constructors:

```
// See also constructors for StrL2d
StrL3d( ) { } // Default constructor.
StrL3d( const P3d &P, const V3d &r ) // Point plus direction.
StrL3d( const P3d &P, const P3d &Q ) // Two points.
```

Definition:

```
// First some methods to define the straight line:
void Def( const P3d &P, const V3d &r );
   // Point plus direction vector.
void Def( const P3d &P, const P3d &Q )
   { Def( P, (V3d) ( Q - P ) ); } // Two points.
```

Operators:

```
friend P3d operator * ( const StrL3d &g, const StrL3d &h );
friend P3d operator * ( const Plane &plane, const StrL3d &s );
void operator = ( const StrL3d &other )
   { Def( other.point, other.dir ); }
Boolean operator == ( const StrL3d &other ) const;
```

Frequently used methods:

```
V3d GetDir( ) const;
   // Return the normalized direction vector r0.
P3d GetPoint( ) const;
   // Return the point that was used for the definition of the line.
void SetPoint( const P3d &P ) { point = P; }
   // Change point that was used for definition of the line.
void SetDir( const V3d &newdir );
   // Change direction vector (newdir need not be normalized)
Real Distance( P3d &P) const;
```

```
    // Return the distance of a point (≥ 0)
P3d InBetweenPoint( Real t ) const;
    // Return the point x⃗ = P⃗ + t r⃗0.
Real GetParameter( P3d &P ) const ;
    // Return the parameter t in the vector equation x⃗ = P⃗ + t r⃗0.
P3d NormalProjectionOfPoint( const P3d &P^ ) const;
    // Return the normal projection of the point dist onto the line.
Boolean ContainsPoint( const P3d &P, Real tol = 1e−4 ) const;
   // Insert the point P in the defining equation of the line
   // and checks whether the result is of absolute value less than tol.
Boolean IsParallel( const StrL3d &other, Real tol = 1e−8 ) const;
    // Check whether the lines are parallel. The angle between the direction
    // vectors must not differ by more than a certain tolerance.
void NormalProjectionOfPoint( const P3d &p, P3d &fp ) const;
    // Project the point p orthogonally onto the line
    // and store the result in fp.
void CommonNormal( const StrL3d &g, V3d &dir,
        P3d &F, P3d &Fg ) const;
    // Determine the common normal of the line and the straight line g.
    // F and Fg are the feet of this line; dir is the
    // direction of the common normal.

// Some methods for the application of transformations.

void Translate( Real dx, Real dy, Real dz );
    // Translate the line by the vector (dx, dy, dz)
void Translate( V3d &v );
    // Translate the line by means of a 3D vector
void Scale( Real kx, Real ky = 0, Real kz = 0 );
    // Apply a scaling operation
void Rotate( StrL3d &a, Real angle_in_deg );
    // Rotate the line about a 3D axis through a given angle (in deg).
void Screw( const StrL3d &a, Real angle, Real parameter );
    // Apply a helical motion defined by axis, angle, and parameter
void Screw( HelicalMotion &S );
    // Apply a helical motion.
void Reflect( const Plane &e );
    // Reflect at a plane

// Drawing routines

void Draw( Color f, Real t1, Real t2, ThinOrThick thick,
        Real offset = STD_OFFSET ) const;
    // Draw the line in color col from the point that corresponds
    // to the parameter t1 to the point  that corresponds to the
    // parameter t2. You have to specify the width of the line:
    // THIN, MEDIUM, THICK, VERY_THICK.
    // When the line is to be drawn on a polygon, one should
    // use an offset (otherwise the z-buffer will have some
    // side-effects), which we will describe later on.
```

```
void LineDotted( Color col, Real t1, Real t2, int n,
      ThinOrThick thick, Real offset = STD_OFFSET ) const;
   // Similar to the ordinary drawing;
   // line dotting with n intervals.
void Dotted( Color col, Real t1, Real t2, int n, Real rad ) const;
   // Similar to the ordinary drawing; draws the line dotted
   // (n "dots", i.e., small circles (pseudo-spheres) with radius rad).
void Dashed( Color c, Real t1, Real t2, int n, Real f,
   ThinOrThick thick, Real offset = STD_OFFSET );
   // Similar to Dotted drawing mode; draws n short dashes.
   // The dash length depends on the real f ∈ [1.01, 3]. The bigger f is,
   // the shorter the dashes are.
```

class *SurfOfRevol*; → declaration in `"revol.h"`

Describes a surface of revolution. Derived from *ParamSurface*.

Constructors:

```
SurfOfRevol( ); // Default constructor.
SurfOfRevol( const StrL3d &a ); // Specify axis.
SurfOfRevol( Color c, Boolean isSolid, int rot_number, Spline3d &k,
      Real w1 = 0, Real w2 = 360 );
   // Define surface via meridian k.
SurfOfRevol( Color c, Boolean solid, int n_vertices,
      Coord3dArray points_on_meridian, int fine, int order,
      Real w1 = 0, Real w2 = 360 );
   // Define surface via an array of points on meridian.
```

Definition:

```
void Def( Color c, Boolean isSolid, int rot_number, L3d &basis, int fine,
      Real w1 = 0, Real w2 = 360,
      const StrL3d &axis = StrL3d( P3d(0,0,0), V3d(0,0,1) ) );
   // rot_number is the number of points on each parallel circle,
   // the L3d object basis will be interpolated by a cubic spline
   // where the number of points on the spline depends on fine.
   // [w1, w2] is the parameter interval for the rotation.
void Def( Color c, Boolean solid, int n_vertices,
      Coord3dArray points_on_meridian,
      int fine, int order, Real w1 = 0, Real w2 = 360,
      const StrL3d &axis = StrL3d( P3d(0,0,0), V3d(0,0,1) ) );
   // Compare preceding definition. Here, the points of the coordinate
   // array will be interpolated.
void Def( Color c, Boolean isSolid, int rot_number, Spline3d &k,
      Real w1 = 0, Real w2 = 360,
      const StrL3d &axis = StrL3d( P3d(0,0,0), V3d(0,0,1) ) );
   // Define surface directly via passing a meridian spline.
```

Additional or overwriting methods:

```
void Shade( FlatOrSmooth smooth, Shininess reflect );
```

```
    // Standard shading method

  // Geometric transformations
  void Translate( Real dx, Real dy, Real dz );
  void Translate( const V3d &v )
  void Scale( Real kx, Real ky = 0, Real kz = 0 );
  void Rotate( const StrL3d &a, Real w );
  void Screw( const StrL3d &a, Real angle, Real parameter );
  void Screw( const HelicalMotion &S );
  void Reflect( const Plane &e );

  void BorderingCircles( Circ3d &k1, Circ3d &k2 );
    // Calculates the bordering circles and stores them in k1 k2.
  void Copy( SurfOfRevol &result ); // Copy method.
```

class *SurfWithThickness*; → declaration in `"paramsurface.h"`

Abstract class derived from *ParamSurface*. Can be used for displaying a parameterized surface with a certain thickness (compare Figure 6.21).

Constructors:

```
SurfWithThickness( P3d (*f0)( Real, Real ) )
  // Constructor needs a pointer to the surface point function.
```

Methods:

```
void ShadeWithThickness( Color c, Real thickness,
                         ThinOrThick outline_style,
                         Boolean contour = true );
  // The only additional method of relevance. c refers to the color
  // of the (usually) small transition between surface and offset.
```

Sample code for better understanding:

```
// Compare "rotoid_surf_with_thickness.cpp".

P3d RotoidSurfacePoint( Real u, Real v )
{
  const Real a = 10, Transmission = 3;
  const StrL3d axis2( P3d( a, 0, 0 ), Ydir );
  P3d P( a + v, 0, 0 );
  P.Rotate( axis2, Deg( Transmission * u ) );
  P.Rotate( Zaxis, Deg( u ) );
  return P;
}

SurfWithThickness Surf( RotoidSurfacePoint );

void Scene::Init( )
```

```
{
    int n = 50;
    Real u1, u2, v1, v2;
    u1 = 0;
    u2 = 2 * PI;
    v1 = -8;
    v2 = 7;
    Surf.Def( Yellow, 4*n, n, u1, u2, v1, v2 );
    Surf.PrepareContour( );
}

void Scene::Draw( )
{
    Surf.ShadeWithThickness( Gray, -0.5, MEDIUM );
}
```

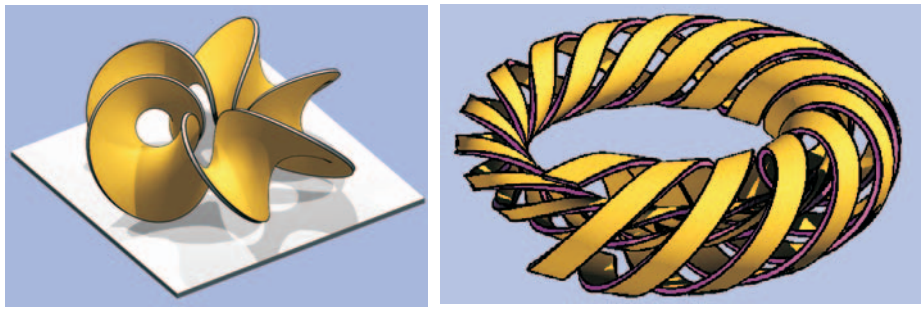

FIGURE 6.21. Two examples for the use of the class *SurfWithThickness*.

class *Torus*; → declaration in `"torus.h"`

Describes a torus with axis z. Derived from *ParamSurface*. Requires inclusion of `"torus.h"` at the top of your file.

Constructor:

Torus( ); // Default constructor.

Definition:

void Def(*Color* c, **int** no_of_parallel_circs, **int** no_of_rotating_circs,
Real radius_of_mid_circ, *Real* radius_of_rot_circ,
Real phi1 = 0, *Real* phi2 = 2 * PI,
Real psi1 = 0, *Real* psi2 = 2 *PI);
// Define torus by specifying number of parallel and meridian
// circles, radii of midcircle, and meridian circle and
// parameter limits.

Additional methods:

```
Real Get_a( ) const; // Return radius of midcircle.
Real Get_b( ) const; // Return radius of parallel circle.
void Shade( FlatOrSmooth smooth = SMOOTH,
            Shininess reflecting = REFLECTING);
  // Shade torus.
```

Sample code for better understanding:

```
// Compare Figure 6.22.

#include "opengeom.h"
#include "defaults3d.h"
#include "torus.h"

Torus Tube;

void Scene::Init( )
{
    Tube.Def( Yellow, 31, 61, 8, 3, -PI, 0 );
    Tube.PrepareContour( );
}
void Scene::Draw( )
{
    Tube.Shade( );
    Tube.WireFrame( Black, 10, 10, THIN );
    Tube.Contour( Black, MEDIUM );
    Tube.DrawBorderLines( Black, MEDIUM, false, true );
}
```

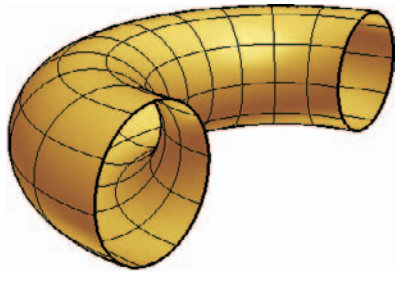

FIGURE 6.22. A half torus.

class *TubSurf*; → declaration in `"tubular.h"`

Describes a tubular surface. Derived from *ParamSurface*.

Methods:

void Def(*Color* c, **int** n, *Coord3dArray* P, **int** fine, *Real* fi1, *Real* fi2, **int** n1, **int** n2, *Real* radius);
// The midline of the surface is a spline through n points defined
// by P. fine determines the number of points on this midline.
// fi1 and fi2 are the parameter limits in direction of the
// characteristic circles. In order to display the complete surface, use
// fi1 = 0 and fi2 = 2 * PI. n1 and n2 are the number of surface
// facets in u and v direction, radius is, of course, the radius of the
// characteristic circles.
CubicSpline3d &Midline( ); // Return the midline of the surface.

class *V3d*; → declaration in `"vector.h"`

Describes a three-dimensional vector (x, y, z). Inherits the methods and operators of *V23d* (partly overwritten!).

Constructors:

V3d( ); // No initialization.
V3d(**const** *V2d* &v); // Initialization with (x0, y0, 0)
// where v = (x0, y0).
V3d(*Real* x0, *Real* y0, *Real* z0); // Initialization with (x0, y0, z0).
V3d(**const** *P3d* &P, **const** *P3d* &Q); // Initialization with $\overrightarrow{PQ}$.

Additional or overwriting operators:

V3d **operator** −( ) **const**;
// Scale vector with −1.
friend *V3d* **operator** + (**const** *V3d* &v, **const** *V3d* &w);
// Vector addition v + w.
friend *V3d* **operator** − (**const** *V3d* &v, **const** *V3d* &w);
// Vector subtraction v − w.
void operator += (**const** *V3d* &v);
// Analog of += operator of ordinary numbers.
void operator −= (**const** *V3d* &v);
// Analog of −= operator of ordinary numbers.
friend *V3d* **operator** * (**const** *Real* t, **const** *V3d* &v);
// Scaling by a real number.
friend *Real* **operator** * (**const** *V3d* &v, **const** *V3d* &w);
// Dot product v · w.
friend *V3d* **operator** ^ (**const** *V3d* &v, **const** *V3d* &w);
// Vector product v × w.

Additional or overwriting methods:

Real XYLength( );
// Return the length of the projection onto $[x, y]$: $\sqrt{x^2 + y^2}$.
Real Slope( ) **const**;
// Return the elevation angle φ of the vector in radians
// $(-\pi/2 < \varphi \leq \pi/2)$.

```
Real SlopeInDeg( ) const;
    // Return the elevation angle φ of the vector in degrees
    // (−90° < φ ≤ 90°).
void Cartesian( Real dist, Real azim_in_deg, Real elev_in_deg );
    // Calculate the Cartesian coordinates of a vector, where the
    // spherical coordinates (length, azimuth, and elevation in degrees)
    // are given.
void Spherical( Real &length,
                Real &azim_in_deg, Real &elev_in_deg ) const;
    // Calculate the spherical coordinates of a vector (length, azimuth,
    // and elevation in degrees) from its Cartesian coordinates x, y, z.
void AssignTo( Vector v );
    // Copy vector to an array double [3]
void Rotate( const V3d &a, Real angle_in_deg );
    // Rotate about a vector.
void Rotate( const RotMatrix &m );
    // Multiply vector by rotation matrix.
void Reflect( const Plane &e );
    // Reflect at a plane.
Boolean Collinear( const V3d &v, Real tol = 1e−4 ) const;
    // Check whether vector is collinear with another vector v.
    // Default tolerance is 1e−4
void OrthoBase( V3d &unit_vec1, V3d &unit_vec2 );
    // Normalize vector and calculate two unit vectors so that
    // the three vectors form an ortho-base and unit_vec1 is horizontal.
    // Note that the vector is changed!
Real Angle( const V3d &v, Boolean already_normalized = false,
    Boolean check_sign = false ) const;
    // Returns the angle between two vectors (in arc length!).
    // Use Deg( Angle(...)) to get the angle in degree.
```

6.3 Many More Useful Classes and Routines

class *GammaBuffer*; → declaration in `"gbuffer.h"`

Describes a function graph that is given over a rectangular domain. The graph is given only numerically by z-values over a fine grid of $1024 \times 1024 \approx 1$ Million (!) grid points. Thus, the surface is actually a huge buffer. In fact, it works like z-buffering: It can be changed ("updated") by buffering polyhedra or other function graphs. The buffer was implemented for the simulation of NC-milling (Figure 6.24). We list only a few member functions and give sample code for them (Figure 6.23). The rest is left to the advanced reader.

Constructors:

```
GammaBuffer( ); // Default constructor.
GammaBuffer( Real x1, Real x2, Real y1, Real y2, Real z1, Real z2 );
```

```
// Define the buffers rectangular domain and extreme z−values
```

Definition:

```
void Def( Real x1, Real x2, Real y1, Real y2, Real z1, Real z2 );
  // See constructor.
void Def( const Box &bounding_box );
  // Domain can also be given by an object of type Box.
```

Methods:

```
void Clear( ); // clear buffer
void PutPolyhedron( Polyhedron &P, int obj_idx );
  // Buffer an entire polyhedron.
void PutSurface( ParamSurface &P, int obj_idx );
  // Buffer an entire surface.
void Show( Color col = Gray, Boolean box = true, int delta0 = 16 );
  // Display the resulting surface in a single color
  // (with or without bounding box).
  // Depending on the complexity, delta0 should vary.
void ShowSurf( int delta0 = 16 );
  // Show surface (multicolored); the more complex
  // the surface is, the smaller delta0 must be.
void ShowBox( Color col = Gray );
  // Show only the bounding box.
void Delete( );
void StoreInFile( const char *name );
void GetFromFile( const char *name );
```

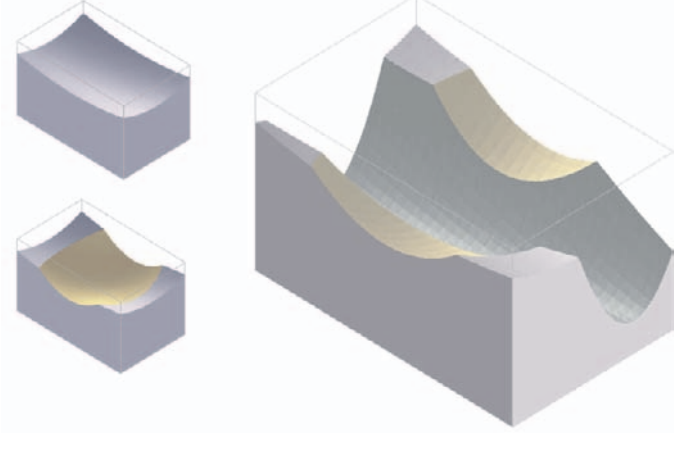

FIGURE 6.23. A typical *GammaBuffer*, i.e., an arbitrary complex function graph that is given numerically (output of the given sample code). Left: Buffer after the first and second update, right: Buffer after the third update.

Sample code for better understanding:

```
#include "opengeom.h"
#include "defaults3d.h"
#include "gbuffer.h"

// define three function graphs
class Graph1: public FunctionGraph
{
   Real z( Real x, Real y ) { return 2 + ( x * x + 0.3 * y * y ) / 20; }
};
class Graph2: public FunctionGraph
{
   Real z( Real x, Real y ) { return -1 + x * x / 10 + y * y / 10 ; }
};
class Graph3: public FunctionGraph
{
   Real z( Real x, Real y ) { return -3 + Sqr( x + 0.2 * y - 1) / 2 ; }
};
Graph1 G1;
Graph2 G2;
Graph3 G3;
GammaBuffer Buffer;
void Scene::Init( )
{
   int n = 30;
   Real x1 = -5, x2 = -x1, y1 = -8, y2 = -y1;
   G1.Def( Gray, n, n, x1, x2, y1, y2 );
   G2.Def( Gray, n, n, x1, x2, y1, y2 );
   G3.Def( Gray, n, n, x1, x2, y1, y2 );
   // put surfaces into buffer
   Buffer.Def( x1, x2, y1, y2, -5, 6 );
   Buffer.Clear( );
   Buffer.PutPolyhedron( G1, 0 );
   Buffer.PutPolyhedron( G2, 1 );
   Buffer.PutPolyhedron( G3, 2 );
}
void Scene::Draw( )
{
   Buffer.Show( Gray, true, 4 );
}
```

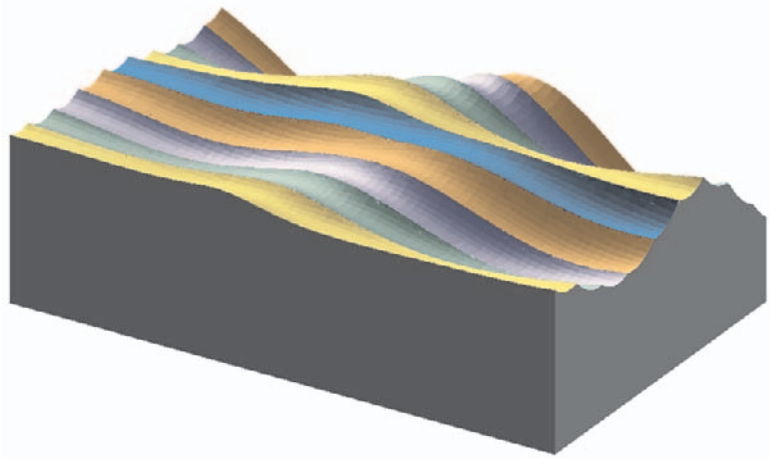

FIGURE 6.24. Another typical *GammaBuffer* (output of `"milling.cpp"`).

struct *IntArray*; → declaration in `"intarray.h"`

"Intelligent" structure[3] that consists of an arbitrary number of integers. This number is accessible as public member variable size, i.e., the members have indices from 0 to size − 1. The structure handles dynamic memory allocation and deletion automatically. CompareSee also page 600.

Constructors:

IntArray( ) // Default constructor.
IntArray(**int** n, **int** init_val = DUMMY)
// Defines size and sets all elements to init_val.
// DUMMY is some weird integer.

Operators:

int & **operator** [ ] (**int** i) **const**
// Access to the *i*-th element of the structure.
void operator = (**const** *IntArray* &other)
// Assign the values of the integer array other to the structure.

Definition:

void Def(**int** n, **int** init_val = DUMMY);
// Defines size and set all elements to init_val.
// DUMMY is some weird integer.

[3]A structure in C++ is the same as a class with no private member variables and member functions.

Methods:

```
void Copy( IntArray &f ) const;
// Assign the values of the integer array f to the structure.
void Sort( int i1 = 0, int i2 = -1, int sgn = 1 );
// Sort elements with indices between i1 and i2 in ascending (sgn= 1)
// or descending (sgn= -1) order. If i2 < i1 (default), i2 is set to
// size-1.

// The following methods return minimal, maximal, and average value.
int Max( ) const;
int Min( ) const;
Real Average( ) const;
```

class *LinearSystem*; → declaration in `"linsyst.h"`

Describes a system of n linear equations in n unknowns and provides methods for its manipulation and solution by means of GAUSS elimination.

Constructors:

```
LinearSystem( ); // Default constructor.
LinearSystem( int n ) // Allocate memory for coefficients.
```

Definition:

```
void Def( int n ); // See constructor.
```

Operators and methods:

```
Real & operator ( ) ( int i, int j );
   // Return the coefficient c[i][j].
   // Caution: Indices i and j start with 1, not 0!
   // The last column (index n + 1) stores the perturbation column.
void SetHorRow( int i, const Real hor[ ] ); // Set horizontal row i.
void SetVertRow( int j, const Real vert[ ] ); // Set vertical row j.
void SetMatrix( const Real *matrix ); // Set all coefficients.
Boolean GetSolution( Real *result );
   // Calculate the solution vector of the system.
   // Return true if successul and false if system is singular.
```

Sample code for better understanding:

```
LinearSystem LinSyst;
LinSyst.Def( 2 );
Real hor[3];
hor[0] = 3, hor[1] = 2, hor[2] = 2;
LinSyst.SetHorRow( 1, hor );
hor[0] = 1, hor[1] = 1, hor[2] = 2;
LinSyst.SetHorRow( 2, hor );
Real *result = NULL;
ALLOC_ARRAY( Real, result, 2, "result" );
if ( LinSyst.GetSolution( result ) )
{
    char str[200];
    sprintf( str, "x =%.5f,y =%.5f", result[0], result[1] );
    ShowString( str );
}
FREE_ARRAY( Real, result, 2, "result" );
```

class *O23d*; → declaration in `"o23d.h"`

Object of 2D or 3D points. Many of OPEN GEOMETRY's geometric classes are derived from this class. It describes a conglomerate of 3D (or 2D) points and provides a few useful methods that apply to *all* derived classes. The classes *O2d* and *O3d* are directly derived from *O23d*. They are the ancestors of many 2D and 3D classes.

Constructors:

O23d( ) // Default constructor.
O23d(*Color* f, **int** numOfPoints) // Specify color and size.

Definition:

void Def(*Color* f, **int** numOfPoints); // Specify color and size.

Operators:

void operator = (*O23d* &other) // Copy one object to another.
P3d &**operator** [ ] (**int** i)
// By writing ObjectName.pPoints[i] you can directly access any
// member point. This is very useful with many derived classes!
// Note that indexing starts with 1 and ends with nPoints.

Methods (common to all derived classes!):

int Size( ) **const** // Return the number of points.
void SetSize(**int** n);
// Set number of objects (only decreasing is permitted).
Color F(*Color* f) // Returns the object color.

```
Color GetColor( ) const
void ChangeColor( Color new_color )
void MarkPoints( Color f, Real r1 = 0.2,
                 Real r2 = 0, int step = 1 );
   // Marking method for object points. A point is marked by a
   // circle of color f and radius r1 and a circle of radius r2 in the
   // background color. If step = 1, every point is marked,
   // if step = 2 every second, etc.
int IsPoint( P3d &X, Real tol = 1e-4 );
   // Return 0 if X is not an element of the object. Otherwise, the
   // index of the corresponding object point is returned.
P3d * FirstPoint( ) // Return first object point.
P3d * LastPoint( ) // Return last object point.
void Copy( O23d &result, int n1 = -1, int n2 = -1 ) const;
   // Copy object points with indices between n1 and n2 to result.
   // By default, all object points are copied.
P23d GetBaryCenter( ) const;
   // "Barycenter" has average coordinates of all points.
   // Works in 2-space and 3-space.
void GetBoundingBox( Box &bbox ) const;
   // Bounding box with axis-parallel sides.
   // In 2-space, the box is flat.
P3d GetCenterOfBoundingBox( ) const;
   // Return the center of the bounding box.
```

Public member variables:

```
P3d *pPoints; // array of pointers to member points
int nPoints;  // number of points
Color col;    // color
```

class *P23d*; → declaration in `"points.h"`

Describes a 2D or 3D point. The class inherits the methods and operators of *V23d*, *V2d* and *V3d* and serves as a base class for *P2d* and *P3d*. It is described in [14], p. 77. Therefore, we simply list its constructors and methods without further comment.

Constructors and Methods:

```
P23d( )
P23d( V23d &v )
void Mark( Color f, Real r1, Real r2 = 0, int Dim = 0 ) const;
Real Distance( const P23d &P) const ;
P23d InBetweenPoint( const P23d &other, Real t ) const;
```

class *Projection*; → declaration in `"camera.h"`

Describes a very versatile "virtual camera" in 3-space. This camera works very much like an ordinary camera, and its methods are named

similarly. For better understanding, please see Section 3.2. A major change from version 1.0 is that the member functions are now virtual. This has no visual effect on the applications; it was, however, necessary for the implementation of multi scene applications.

Definition:

```
virtual void Def( );
      // This member-function has to be implemented by the user!
virtual void Def( Real eye_x, Real eye_y, Real eye_z,
          Real target_x, Real target_y, Real target_z,
          Real fovyAngle = 20 );
      // fovy = field of vision angle ( 1 < fovy < 65 )
virtual void DefaultCamera( Real eye_x, Real eye_y, Real eye_z );
      // Perspective: eye( x, y, z ), target(0, 0, 0 ),
      // fovy = 20 (focus = 50).
virtual void DefaultOrthoProj( Real eye_x, Real eye_y, Real eye_z );
      // Works like DefaultCamera( ), but switches to ortho projection.
```

Methods:

```
// Change camera lens (zoom in and out with factor k)
   virtual void ZoomIn( Real k );
       // k = 1 ... no effect; k = 1.05 ... 5% magnification etc.
   virtual void ZoomOut( Real k );
   virtual void ChangeFocus( Real focus_in_deg );
       // Default is 50 ... ordinary camera lens
       // 15 <= focus <= 35 ... wide-angle
       //  focus >= 70 ... teleobjective lens
// Change eye point:
   virtual void ChangePosition( const P3d &NewEye );
   virtual void ChangePosition( Real eye_x, Real eye_y, Real eye_z );
// Rotate camera (target unchanged):
   virtual void RotateVertically( Real angle_in_deg );
       // Rotate about z−parallel axis through target point.
   virtual void RotateHorizontally( Real angle_in_deg );
   virtual void AutoRotation( Boolean switch_on );
       // Starts or stops vertical autorotation (1 degree per frame).
// Change target point:
   virtual void ChangeTarget( const P3d &NewTarget );
   virtual void ChangeTarget( Real target_x, Real target_y,
                                                Real target_z );
// Rotate camera (eye unchanged):
   virtual void TargetLeftRight( Real angle_in_deg );
   virtual void TargetUpDown( Real angle_in_deg );
// Move camera (change eye and target point):
   virtual void Translate( Real dx, Real dy, Real dz );
   virtual void Forward( Real translation );
   virtual void Backward( Real translation );
   virtual void UpDown( Real translation );
   virtual void LeftRight( Real translation );
```

```
// Twist camera:
    virtual void Roll( Real angle_in_deg );
    virtual void ChangeTwist( Real angle_in_deg );
// Special views:
    virtual void TopView( );
    virtual void FrontView( );
    virtual void RightSideView( );
    virtual void SpecialView( char c );
        // '1' = top view, '2' = front view, '3' = right side view
        // '4' = bottom view, '5' = back view, '6' = left side view
// Switch from perspective to orthoprojection and vice versa:
    virtual void SwitchProjection( );
    virtual void SwitchToNormalProjection( );
// Control depth buffering:
    virtual void Zbuffer( Boolean on_off );
    virtual void ClearZBuffer( );
    virtual void SetDepthRange( Real z_min, Real z_max );

// Get data from class (all angles are returned in degrees):
    virtual Real GetAzim( ) const;
    virtual Real GetElev( ) const;
    virtual Real GetFocus( ) const;
    virtual Real GetFovy( ) const;
    virtual Real GetTwist( ) const;
    virtual P3d GetPosition( ) const;
    virtual P3d GetTarget( ) const;
    virtual Plane GetProjPlane( ) const;
    virtual Real GetDist( ) const;
    virtual Boolean IsActive( ) const;
    inline Boolean IsOrthoProj( ) const;
    virtual Boolean IsSpecialView( ) const;
    virtual void GetDepthRange( Real &z_min, Real &z_max );
// Save data to file 'tmp.dat':
    virtual void SaveEyeAndLight( );
// Change Light:
    virtual void ParallelLight( Real x, Real y, Real z );
    virtual void SetLocalLight( Real x, Real y, Real z );
// Intersection of the proj. ray with the image plane:
    virtual P3d ProjectPoint( const P3d &P );
// Pixel coords in the current window:
    virtual P3d CoordsInScreenSystem( const P3d &P );
        // The z-coord. of the result is a depth value (for z-buffering).
    virtual P3d CoordsInProjPlane( const P3d &P );
// Projection ray in direction eye point:
    virtual V3d GetProjRay( const P3d &P ) const;
    virtual V3d GetNormalizedPrincipalRay( ) const;
// Make light heliocentric instead of geocentric:
    virtual void Heliocentric( Boolean on_off );
// Save and restore camera temporarily:
```

```
    virtual void Save( );
    virtual void Restore( );
// Some more member functions:
    virtual V3d GetProjDir( ) const;
        // Return the normalized projection vector for -orthoprojection.
    virtual void Configure( Boolean calc_eye );
    virtual void SetClipDist( Real clipdist );
    virtual Boolean ZbufferIsActive( );
    virtual Projection & operator = ( const Projection & );
        // Allow to copy to another camera.
```

class *PulsingReal*; → declaration in `"pulsing.h"`

Describes a real number that changes its value from a minimum to a maximum, starting with some intermediate number. The change is described by an increment, and it can be *LINEAR* or *HARMONIC*. This class is very useful for animations.

Constructors:

PulsingReal( ) { value = DUMMY; }

There is only the default constructor and only one way of defining an instance of *PulsingReal*.

Definition:

```
void Def( Real init_val, // the init value
          Real incr,  // the first increment
          Real max_val, // the maximum value
          Real min_val, // the minimum value
          HowToPulse how ); // LINEAR or HARMONIC
```

Methods and operators:

```
Real operator ( ) ( void ) { return value; }
Real Next( ); // Compute next value.
void Scale( Real factor );
```

Sample code for better understanding:

```
PulsingReal Pulse;
const Real init_val = 5, incr = 0.02, min = 2, max = 8;
Pulse.Def( init val, incr, min, max, HARMONIC );
Real a = Pulse.Next( );
Real b = Pulse( ); // a and b have the same value
```

class *RandNum*; → declaration in `"randnum.h"`

Generates random numbers of type *Real*. Base class.

Constructors, operators and methods:

```
RandNum( ); // Default constructor.
RandNum( Real min, Real max );
    // Generate GAUSS distributed random numbers in [min, max].
void Def( Real min, Real max ); // See constructor.
Real operator ( ) ( void ); // Operator to get a random number.
Real Next( ); // Method to get a random number.
```

struct *RealArray*; → declaration in `"realarray.h"`

"Intelligent" structure that consists of an arbitrary number of reals. This number is accessible as public member variable size, i.e., the members have indices from 0 to size − 1. The structure handles dynamic memory allocation and deletion automatically. Compare also page 600.

Constructors:

```
RealArray( ) // Default constructor.
RealArray( int n  ) // sets size to n
```

Operators:

```
Real & operator [ ] ( int i ) const
// Access to the i^th element of the structure.
void operator = ( const RealArray &other )
// Assign the values of the real array other to the structure.
```

Definition:

```
void Def( int n, Real init_val = DUMMY );
// Define size and set all elements to init_val.
// DUMMY is some weird integer.
```

Methods:

```
void Copy( RealArray &f ) const;
// Assign the values of the real array f to the structure.
void Sort( int i1 = 0, int i2 = −1, int sgn = 1 );
// Sort elements with indices between i1 and i2 in ascending (sgn= 1)
// or descending (sgn= −1) order. If i2 < i1 (default), i2 is set to
// size−1.

// The following methods return minimal, maximal, and average value.
int Max( ) const;
int Min( ) const;
Real Average( ) const;
```

class *Scene*; → declaration in `"scene.h"`

A very basic class in OPEN GEOMETRY. Essentially, every OPEN GEOMETRY program derives a successor from *Scene* by implementing the virtual member functions Init( ), Draw( ), CleanUp( ), and Animate( ).

Virtual member functions:

```
virtual void Init( );
virtual void Draw( );
virtual void CleanUp( );
virtual void Animate( );
```

Methods:

```
void NoOffset( ); // Sets the global offset for z-buffering to 0.
void Offset( Real delta = STD_OFFSET );
   // Sets the global offset for z-buffering to delta.

// Drawing methods for coordinate systems in 2D and 3D:
void ShowAxes3d( Color f, Real sizex,
      Real sizey = 0, Real sizez = 0, Boolean show_xyz = true );
void ShowAxes( Color f, Real sizex,
      Real sizey = 0, Real sizez = 0 );
   // Do not use this method in new programs. It is onlyvalid
   // only in order to stay compatible with version 1.0.
void ShowAxes3d( Color c,
                 Real xmin, Real xmax,
                 Real ymin, Real ymax,
                 Real zmin, Real zmax, Boolean show_xyz = false );
void ShowAxes2d( Color c, Real xlength,
                 Real ylength = 0, Boolean show_xy = true );
void ShowAxes2d( Color c,
                 Real xmin, Real xmax,
                 Real ymin, Real ymax, Boolean show_xy = false );

void Zbuffer( Boolean on_off ); // Switch z-buffer on or off.
```

Methods to be called in Init( ):

```
void AllowRestart( Boolean yes_no = true );
   // Allow restart of the program.

void CreateNewPalette( int idx, int size, Real r1, Real g1, Real b1,
                                          Real r2, Real g2, Real b2 );
   // Create your own color.

// Define your own background.
void SetBackgroundColor ( Real r, Real g, Real b );
void SetBackgroundTexture( char *bitmapName, Real alpha = 1 );
```

Methods to be called in Animate( ):

```
// Export utilities:
void PreparePovRayFile( const char *name_of_file );
int SaveAsPovRayFile( const char *filename );
int SaveAsBitmapAnimation( const char *path = "BMP",
                           int start_frame = 1,
                           int increase = 1,
                           int max_files = 50,
                           int bit_depth = 24 );

void Restart( int delay = 0, Boolean restore_camera = false );
void FreezeAnimation( );
Boolean RestartPossible( );
   // Check whether AllowRestart( ) has been called in Init( ).
```

class *SmartText*; → declaration in `"smarttext.h"`

The class *SmartText* allows you to position, display, and manipulate text in an OPEN GEOMETRY window. It can be used to decorate OPEN GEOMETRY presentations with special effects, to display a scrolling text on the screen, etc. Note, however, that displaying a beautifully formatted text slows down the animation. If you do not really need it, you should consider the function PrintString(...) as well. You must include `"smarttext.h"` in order to use *SmartText*.

Constructors:

```
SmartText( ); // Default constructor.
```

Definition:

```
void Def( int n, const P3d &P = Origin,
   const Plane &p = XYplane );
// n is the maximum number of lines, P is the upper left corner,
// and p the supporting plane of the text.
```

Frequently used methods and operators:

```
void SetLine( int n, Color col, Real size, Real lineskip,
const char *txt );
   // Insert the character txt in line number n. lineskip determines the
   // distance between this line and line number n+1, col the text color,
   // and size the text size.
void Display( Real opacity = 1, Boolean TwoD = true );
   // Set opacity and decide whether to display text on the 2D
   // computer screen or on a 3D plane. The text will be put
   // into XYplane and can be transformed by one of the
   // following methods:
void Translate( const V3d &v );
void Rotate( Real angle_in_deg );
void Scale( const Real t );
void SetCorner( const P3d &C );
void ClearText( ); // clears all text lines
```

Sample code for better understanding:

```
#include "opengeom.h"
#include "smarttext.h"

SmartText Text;

void Scene::Init( )
{
  Text.Def( 10 ); // a maximum of 10 lines
  Text.SetCorner( P3d( 0, 10, 0 ) );
  Text.SetLine( 1, Red, 3, 2, "Open Geometry");
  Text.SetLine( 2, Green, 1.5, 1, "This is line 2");
  Text.SetLine( 3, Black, 1.5, 1, "and this,finally,is line 3");
  Text.SetLine( 4, Black, 1.5, 1, "We add line 4");
  Text.SetLine( 5, Black, 1.5, 1, "and line 5 (just in order");
  Text.SetLine( 6, Black, 1.5, 1, "to see how slow we are)");
  ScaleLetters( 0.8, 1 );
}
void Scene::Draw( )
{
  ShowAxes( Blue, 10 ); // Show the axes
  Text.Display( );
}
void Scene::Animate( )
{
  if ( FrameNum( ) % 180 < 90 )
    Text.Translate( V3d( 0, -0.05, 0 ) );
  else
    Text.Rotate( 2 );
}
```

class *V23d*; → declaration in `"vector.h"`

The common ancestor of the classes *V2d* and *V3d* (vectors of dimension two and three, respectively). Besides the operators and methods below, it has three private member variables x, y and z that can be directly accessed. It has been described in [14], pp. 73–74. Since only Print( ) has changed, we list its components without further comment.

Constructors:

V23d( );
V23d(*Real* x0, *Real* y0, *Real* z0 = 0);

Operators:

void operator ( ) (*Real* x0, *Real* y0)

```
void operator ( ) ( Real x0, Real y0, Real z0 )
void operator *= ( Real t );
void operator /= ( Real t );
Boolean operator == ( const V23d &v ) const;
Boolean operator != ( const V23d &v );
void operator = ( const V23d &v );
void operator = ( Vector v );
// Note that the operator -( ) const (scaling with -1) no longer
// exists. It has moved to V2d and V3d, respectively.
```

Methods:

```
void Print( char *str = "@", int dim = 3, int no_of_digits = 3,
   char *add_str = "\n" ) const;
   // Displays the components as a message: 'text = ( x, y, z )'.
   // In case of dim = 2 the output is 'text = ( x, y )',
   // no_of__digits determines the output precision.
void Normalize( V23d &norm ) const;
void Normalize( );
```

6.4 Useful Functions

Open Geometry 1.0 provided a large number of useful functions concerning mathematical operations, screen output, changing of global settings, etc. This library has been enlarged by a few more routines in Open Geometry 2.0. We list them here together with some comments. Since many of them are described in [14] or have been given self-explanatory names, we will not always go into detail.

Some Additional Mathematical Functions

Open Geometry's mathematical functions enlarge the standard functions of the C mathematics library. Some are unknown to C; some return the same result as the corresponding C function but with additional features. For example, Open Geometry's Sqrt(x) returns the square root of x, but unlike sqrt(x), checks whether x is nonnegative. Using this function (or others with built-in security check) can help in avoiding hard-to-find errors (compare Example 6.5)!

Listing from `"H/mathfunc.h"`:

```
inline Real Sqr( Real x ) // square of x
inline int Signum( const Real x ) // sign +1, 0 or −1
inline Real Sqrt( Real x ) // square root with security check
inline Real Tan( Real x ) // tangent function with security check
inline Real Log( Real x ) // logarithmic function with security check
inline int Round( Real x ) // closest integer
inline Real Arc( Real degrees ) // arc length of an angle
inline Real Deg( Real arc ) // angle in degrees

// Arc functions with security check:
inline Real ArcSin( Real x )
inline Real ArcCos( Real x )
inline Real ArcTan2( Real y, Real x ) // returns a value in [−PI/2, 3PI/2[

// Min and Max functions, both for integer and Real variables
inline Real maximum( Real a, Real b )
inline Real minimum( Real a, Real b )
inline int maximum( int a, int b )
inline int minimum( int a, int b )

inline Real CubicRoot( Real x )
Real AreaOfTriangle( const V2d &A, const V2d &B, const V2d &C );

void Tripod( V3d &a, V3d &b, V3d &n );
   // Calculate an ortho base with n/|n| as third vector.
Real Integral( Real (*f)( Real ), Real t1, Real t2, int n = 100 );
   // Compute ∫_{t1}^{t2} f(t)dt by means of the Simpson formula

// Root finders for algebraic equations:
int QuadrEquation( Real a, Real b, Real c, Real &x1,
   Real &x2, Real eps = 1e−14 );
int ZerosOfPoly3( const Real coeff[4], Real zero[3] );
int ZerosOfPoly4( const Real coeff[5], Real zero[4] );
```

Some of the functions need a closer explanation. The line

```
void Tripod( V3d &a, V3d &b, V3d &n );
```

is equivalent to n.OrthoBase(h, f). After this function is called, n will be normalized, a will be a horizontal vector perpendicular to n and $b = n \times a$.

The use of the integral function will be clear after a look at the following sample code from "simpson.cpp":

```
Real TestFunction( Real t )
{
   return 1 / ( 1 + t * t );
}

int number_of_nodes = 200;
Real t1 = 0, t2 = 1;
Real pi = 4 * Integral( TestFunction, t1, t2, number_of_nodes );
```

QuadrEquation(...) returns the number of real solutions of the equation $ax^2 + bx + c = 0$. The solutions are stored in *ascending* order in x1 and x2. If the solutions are conjugate complex, the real part is stored in x1, the imaginary part in x2.

The functions ZerosOfPoly3(...) and ZerosOfPoly4(...) calculate all *unique non-imaginary* roots of the supplied polynomial. The coefficients of the polynomials have to be put into the *Real* array coeff in order of increasing powers. The roots are stored in the zero array in ascending order; the number of roots is the return value of the function.

Commands for text output

Sometimes it is useful to open a little message window, informing the user about a certain action or displaying the content of a variable. Or perhaps you want to print some text or values directly on the screen. These tasks can be done conveniently by means of Open Geometry routines. You will find a comprehensive listing in "useful.h".

We distinguish between routines that open a message window (alert) and routines that print their output directly on Open Geometry's graphcis screen (compare also [14], p. 66). The following code lines stem from "show_string.cpp". All relevant output methods are used:

```
char str[200];
int n = 20;
Real u = 3;
```

```
sprintf( str, "Here is the result:%.5f/%d =%.5f\n"
   "An integer division may have weird results:%d/%d=%.5f",
   u, n, u / n, (int) u, n, (double) ((int) u / n ) );
ShowString( str );
ShowInteger( "n =", n );
HiAccuracyOutput( );
ShowReal( "u =", u );
SafeExit( );
```

You have output utilities for arbitrary strings, for integers and for reals. Start the program, see how it works, and compare with the code.[4]

Very useful is the SafeExit( ) function. It can be used to prevent undefined operations (e.g., division by zero). Note that you may as well specify a user defined text:

```
SafeExit( "Division by zero!" )
```

We frequently use SafeExit( ) for debugging our code. You can tell at once whether a certain routine is called by inserting a safe exit in its body. Note further that a transcript of all messages is written to the file "try.log" in OPEN GEOMETRY's standard output directory. You can change the output directory by editing "og.ini". This may be necessary if you want to install OPEN GEOMETRY in a network: The output directory requires write permission.

The printing of text on the screen relies almost entirely on the function

```
void PrintString( Color col, Real x, Real y,
                  char *str_with_possible_format_controls,
                  Real var_of_any_type_according_to_format_controls = 1,
                  ...
                  );
```

It allows a character variable to be printed in arbitrary color at an arbitrary position on the screen. Compare the output of "print_string1.cpp" for a test:

Listing from "print_string1.cpp":

```
int n = 20;
Real u = 3;
PrintString( Green, -14, 0, "Here is the result:%.5f/%d =%.5f\n"
   "An integer division may have weird results:%d/%d=%.5f",
   u, n, u / n, (int) u, n, (double) ((int) u / n ) );
```

[4]There exists a further output method named Write(...). It does the same as ShowString(...) but provides a few additional features. They are, however, rather special, and we believe that you won't need them.

In "print_string1.cpp" you can watch a little bug: Move the OPEN GEOMETRY window to the right with your arrow keys. Together with the first letter the whole output vanishes from the screen. Now press <Ctrl+Q> to switch to high quality output. The text reappears. Zooming in a little, you should be able to read it. We believe that the bug is not within our reach. Probably, it can be found somewhere in the OPENGL code.

The combination of PrintString(...) with the following functions yields beautifully formatted text:

```
void HiQuality( Boolean on_off = true );
void ScaleLetters( Real tx, Real ty = 0 );
void InclineLetters( Real degrees );
void RestoreLetters( );
```

HiQuality( ) is called by the shortcut <Ctrl+Q>. It allows you to switch between normal output and high-quality output. The high quality letters are not just simple lines but *triangulated letters* of a roman font type (Too many of them, however, will slow down your code!). The letter size and inclination may also be chosen. If you have ruined the original letters by scaling and inclining, you can restore them by RestoreLetters( ). A simple file to test these methods is "print_string2.cpp".

A still more advanced output method is WriteNice(...). For 2D drawing this routine is similar to the PrintString(...) routine. Additionally, it allows one to rotate the text and to set an α-value for transparency ($0 \leq \alpha \leq 1$, default is $\alpha = 1$). In a 3D window it allows you to write text on an arbitrary plane. Its syntax is

```
void WriteNice( Color col, char *Text,
                Real x, Real y, Real rot_deg, Real size,
                Real opacity = 1, const Plane &TextPlane = XYplane );
```

The variables x, y, and rot_deg determine the position of the text with respect to a certain coordinate system in the plane. The origin is some point in the plane, usually the first point mentioned in Def(...) or in the plane constructor. The x-axis is horizontal; the y-axis is perpendicular to x. A simple example of the use of WriteNice(...) is "hello_world.cpp".

Listing from "hello_world.cpp":

```
#include "opengeom.h"
#include "defaults3d.h"

Plane p;
```

```
void Scene::Init( )
{
   p.Def( P3d( 0, 0, 3 ), V3d( 0, 1, 1 ) );
   InclineLetters( 60 );
}

void Scene::Draw( )
{
   WriteNice( Blue, "Open Geometry", −8, 0, 0, 4, 1, p );
}
```

Note that you can also print special characters such as $ or Greek letters. You cannot use them directly as arguments but you can insert the following in WriteNice(...):

`$Dollar$`, `$alpha$`, `$beta$`,`$gamma$`, `$delta$`, `$eta$`, `$iota$`, `$kappa$`,

`$epsilon$`, `$mu$`, `$nu$`,`$pi$`, `$theta$`, `$chi$`, `$rho$`, `$sigma$` etc.

They will show as

$, α, β, γ, δ, η, ι, κ, ε, μ, ν, π, θ, χ, ϱ and σ.

Capital Greek letters begin with a capital Latin letter:

`$Omega$`, `$Psi$`, `$Lambda$`, `$Xi$`, etc.

will produce the output

Ω, Ψ, Λ, and Ξ,

respectively.

How to show text files and source code

A text file can be displayed very simply by means of the command

```
void ShowTextFile( const char *name );
```

The program then opens a window and displays the contents of the file.

Open Geometry additionally offers the possibility of displaying the source code of a program. The two routines

```
void SetSource( const char *name, const char *first_text );
void ShowSource( );
```

have to be called. You know this from the demo executables in the `"DEMOS/"` directory.

Sample code for better understanding:

```
#include "opengeom.h"
#include "defaults2d.h"

void Scene::Init( )
{
  SetSource( "HANDBOOK/display_code.cpp", "Show source" );
}

void Scene::Draw( )
{
  if ( FrameNum( ) == 1 )
    ShowSource( );
}
```

In the above sample program you would not explicitly have to call the ShowSource( ) function. When the file name is set with the SetSource(...) function, the code can be viewed at any time by means of the key combination <Alt+s> or the menu item "Program→Show source code".

Methods for drawing and shading

Among the most frequently used functions in Open Geometry are the commands *StraightLine2d*(...) and *StraightLine3d*(...). Every time you draw a straight or curved line one of them is called.[5] You may call them according to the following syntax:

void StraightLine2d(*Color* c, **const** *P2d* &P, **const** *P2d* &Q,
ThinOrThick thick)
void StraightLine2d(*Color* c, *Real* x1, *Real* y1, *Real* x2, *Real* y2,
ThinOrThick thick)

void StraightLine3d(*Color* c, **const** *P3d* &P, **const** *P3d* &Q,
ThinOrThick thick, *Real* offset)

As a result, a straight line segment in the specified color and line style will be drawn. It connects the points P and Q (or (x1, y1) and (x2, y2)).

The *StraightLine* functions have existed for a long time, while another useful command has been introduced only recently: MarkAngle(...). It draws a circle

[5] "Curved" lines in Open Geometry consist of a sufficient number of straight line segments.

segment to mark the angle between two straight lines. Additionally, it returns the angle in degrees. Its syntax is

```
Real MarkAngle( Color c,
                const P2d &FirstPoint,
                const P2d &CenterOfAngle,
                const P2d &LastPoint,
                Real radius,
                int no_of_points, ThinOrThick style );
```

or

```
Real MarkAngle( Color c,
                const P3d &FirstPoint,
                const P3d &CenterOfAngle,
                const P3d &LastPoint,
                Real radius,
                int no_of_points, ThinOrThick style,
                Real offset = STD_OFFSET );
```

Note that LastPoint need not really be the arc's end point. It is enough that it be situated on the connecting line of center and end point.

Open Geometry allows you to equip any object with a certain transparency. In order to shade a polygon poly with transparency, you have to insert

```
SetOpacity( 0.5 )
poly.Shade( );
SetOpacity( 1 ) // restores default opacity
```

in Draw( ). The argument of SetOpacity(...) (the α-value) may range from 0 to 1, where $\alpha = 1$ is the default value and means "no transparency," and $\alpha = 0$ makes the object completely invisible. You can get the current value by calling GetOpacity( ).

Be aware that SetOpacity(...) depends on the drawing order! The above code lines effect transparency only with respect to those objects that were drawn or shaded *before* poly. That is, you should put them at the end of Draw( ). Compare Figure 6.25 and the output of the sample file `"opacity.cpp"`:

Listing from `"opacity.cpp"`:

```
Circ3d C[3], D[3]; const Color Col[3] = { Red, Green, Blue }; void

Scene::Init( )
{
   int i;
   for ( i = 0; i < 3; i++ )
   {
      C[i].Def( Col[i], P3d( -6, 0, i ), Zdir, 5, 50, FILLED );
      D[i].Def( Col[i], P3d( 6, 0, i ), Zdir, 5, 50, FILLED );
   }
}

void Scene::Draw( )
{
   SetOpacity( 0.5 );

   // correct drawing order
   C[0].Draw( THICK );
   C[1].Draw( THICK );
   C[2].Draw( THICK );

   // wrong drawing order
   D[2].Draw( THICK );
   D[1].Draw( THICK );
   D[0].Draw( THICK );

   SetOpacity( 1 );
}
```

FIGURE 6.25. Correct and incorrect use of SetOpacity(...). The circles on the right-hand side are drawn with opacity but in incorrect drawing order (output of `"opacity.cpp"`).

A further method for achieving special output effects is *CreateAura*(...). A good example of its usage is `"two_circles.cpp"` (Figure 6.26). Two circles and a couple of straight lines are displayed. In order to improve the 3D effect of the image, we create a white aura around certain objects. You have to call *CreateAura*(...) immediately before the respective drawing routine and you must pass a line width as parameter:

Listing from `"two_circles.cpp"`:

```
Circ3d C1, C2;
P3d H;
StrL3d Axis2;

void Scene::Init( )
{
    ScaleLetters( 1.5 );
    H( 0, 5, 0 );
    C1.Def( Black, H, Zaxis, 200 );
    Axis2 = Xaxis;
    Axis2.Translate( H );
    C2.Def( Black, Origin, Axis2, 200 );
}

void Scene::Draw( )
{
    ShowAxes( Black, 6.5 );
    CreateAura( VERY_THICK );
    Zaxis.LineDotted( Black, -6, 6, 20, MEDIUM );
    CreateAura( VERY_THICK );
    Axis2.LineDotted( Black, -6, 6, 20, MEDIUM );
    CreateAura( VERY_THICK );
    C1.Draw( MEDIUM );
    CreateAura( VERY_THICK );
    C2.Draw( MEDIUM );
    Origin.Mark( Black, 0.1, 0.05 );
    H.Mark( Black, 0.1, 0.05 );
}
```

Note that on some (not on all!) engines the use of auras slows down the animation. You can disable all auras by simply inserting

```
AllowAura( false );
```

in Init( ) or Draw( ).

In Figure 6.26 you can see Open Geometry's standard 3D coordinate system. In Open Geometry 1.0 this was done by calling

```
ShowAxes ( Color f, Real xsize, Real ysize= 0, Real zsize= 0 );
```

This drawing method still exists in Open Geometry 2.0, but we would be happy if from now on you used the following instead:

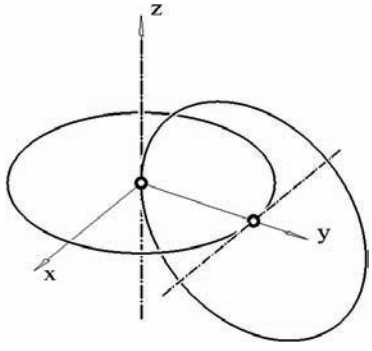

FIGURE 6.26. Using *CreateAura*(...) for achieving special 3D effects (output of `"two_circles.cpp"`).

Listing from `"H/scene.h"`:

```
void ShowAxes3d( Color c, Real sizex,
      Real sizey = 0, Real sizez = 0, Boolean show_xyz = true );
void ShowAxes3d( Color c,
                 Real xmin, Real xmax,
                 Real ymin, Real ymax,
                 Real zmin, Real zmax, Boolean show_xyz = false );
void ShowAxes2d( Color c, Real xlength,
                 Real ylength = 0, Boolean show_xy = true );
void ShowAxes2d( Color c,
                 Real xmin, Real xmax,
                 Real ymin, Real ymax, Boolean show_xy = false );
```

The meaning of the input parameters is not difficult to understand. In both 2D and 3D you have two possibilities:

- Starting from the origin, you can draw all axes with a certain length (sizex, xlength, ...). To make it simple, it is permissible to specify only one length, which will result in equal axis length.
- You can draw all axes in a specified interval ([xmin, xmax], ...).

At the positive end of each axis a little arrow will be attached. If you like, you can also equip them with the letters x, y, and z. To give you an example, the lines

```
ShowAxes2d( Black, -5, 5, -1, 5, true );
```

ShowAxes2d(*Black*, 8, 5, false);
ShowAxes3d(*Black*, 10);

result in the output displayed in Figure 6.27 (from left to right).

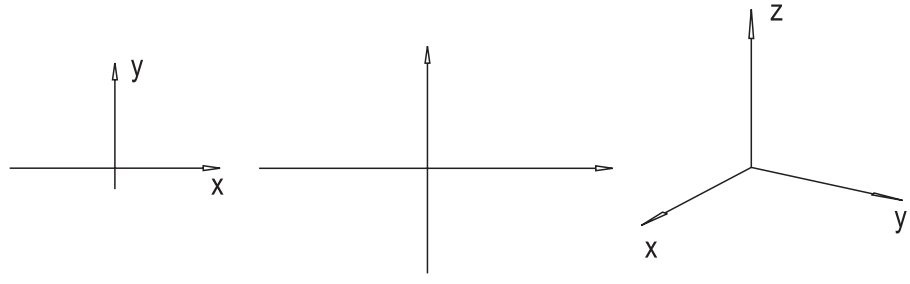

FIGURE 6.27. Different options for displaying coordinate systems in Open Geometry 2.0.

In Open Geometry you have a number of predefined colors. You can address them by a predefined enumeration type named *Color* (defined in `"enums.h"`). For your convenience we list them here as well:

White, AlmostWhite, PureWhite, Black

Blue, *Brown*, *Cyan*, *Gray*, *Green*, *Magenta*,
Orange, *Pink*, *Red*, *Yellow*

DarkBlue, *DarkBrown*, *DarkCyan*, *DarkGray*, *DarkGreen*,
DarkMagenta, *DarkOrange*, *DarkPink*, *DarkRed*, *DarkYellow*

LightBlue, *LightBrown*, *LightCyan*, *LightGray*, *LightGreen*,
LightMagenta, *LightOrange*, *LightPink*, *LightRed*, *LightYellow*

// Independent of eye point and illumination,
// pure colors will always be plotted in one hue.
PureBlue, *PureBrown*, *PureCyan*, *PureGray*, *PureGreen*,
PureMagenta, *PureOrange*, *PurePink*, *PureRed*, *PureYellow*

// Additional types for special purposes.
NoColor, *NotSpecified*

When shading a polygon, Open Geometry takes into account the viewpoint and the light source. Thus, a red polygon may be shaded in different hues according to the camera and light settings you choose. If you do not want this you should use *PureRed* instead. This color provides only one hue.

You can, of course, create your own colors as well. In order to do this you have to insert something like

```
Color MyColor = NewColor( 1 );

const int palette_size = 100;
const int r1 = 0.1, g1 = 0.2, b1 = 0.4;
const int r2 = 0.95, g2 = 0.95, b2 = 1;
CreateNewPalette( MyColor, palette_size, r1, g1, b2, r2, g2, b2 );
```

in Init( ). *NewColor*(...) is an Open Geometry macro that initializes MyColor. *CreateNewPalette*(...) takes eight parameters: It generates a palette of palette_size different hues ranging from the two colors with RGB values (r1, g1, b1) and (r2, g2, b2). If you want to create a pure color, you can use palette_size = 2 and r2 = r1, g2 = g1, b2 = b1. After having created your new palette, you can use MyColor just as any other Open Geometry color.

6.5 Do's and Dont's

This section's intention is to give you a few hints how to make full use of Open Geometry's abilities and to point out typical mistakes. We have already tried to do this in describing Open Geometry classes, but one thing or the other might have been forgotten or had only minor impact on the reader's consciousness. Some of the blunders stem from reader requests after the publication of [14]. Most of them, however, come from our own experience. We will illustrate them with various examples and point out the main problems.

Parameterized curves and surfaces

The typical way to display a plane curve c in Open Geometry is the use of the purely virtual class *ParamCurve2d*. The mathematical equivalent to this class is the *parametric representation* of c, e.g., a function

$$\vec{x}\colon I \subset \mathbb{R} \to \mathbb{R}^2, \quad t \mapsto \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}.$$

Alternatively, you can define c as the graph of a function $f\colon I \subset \mathbb{R} \to \mathbb{R}$, e.g., the set of all points (x, y) satisfying $y = f(x)$. However, this is just the special parametric representation $\vec{x}(t) = \big(t, f(t)\big)$.

The parametric representation of c is never uniquely determined. By applying a *regular parametric transformation* $t = t(t^*)$, one gets a new parameterization $\vec{x}^*(t^*) = \vec{x}\big(t(t^*)\big)$. For a mathematician, the new function $\vec{x}^*(t^*)$ is equivalent to $\vec{x}(t)$. In computer graphics, this is *not true*! We will demonstrate this in an example:

Example 6.1. Bad parametric representation
In `"bad_par_rep.cpp"` we display three differently parameterized copies of a circle catacaustic c (compare Example 2.6).

The best parametric representation uses trigonometric functions and reads

$$x(\varphi) = \frac{-er\big(r - e(2\sin^2\varphi + 1)\cos\varphi\big)}{r^2 - 3er\cos\varphi + 2e^2},$$
$$y(\varphi) = \frac{2e^2 r\sin^3\varphi}{r^2 - 3er\cos\varphi + 2e^2}.$$

By $\varphi(u) = \arccos u$, we transform the φ-parameter interval $[-\pi, \pi]$ to $[-r, r]$ and get

$$x(u) = \frac{r^4 + 3er^2u - 2eu^3}{r^4 - er^2(3u - 2e)},$$
$$y(u) = \frac{2e(r^2 - u^2)^{3/2}}{r^4 - er^2(3u - 2e)}.$$

By substituting

$$\cos\varphi = \frac{1 - t^2}{1 + t^2}, \qquad \sin\varphi = \frac{2t}{1 + t^2},$$

or equivalently by transforming $\varphi(t) = 2\arctan t$, we get a rational parameterization $\vec{x}(t) = \big(x(t), y(t)\big)^t$ of c. The parameter t varies in $(-\infty, \infty)$. Since $\vec{x}(t)$ is rather long, we omit displaying it here.

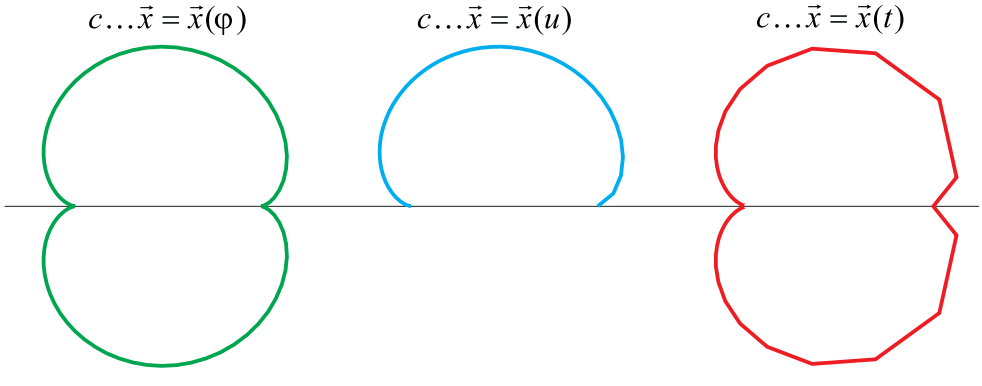

FIGURE 6.28. Three different parametric representations of the same curve.

The effects of the different parametric representations can be judged by the output of `"bad_par_rep.cpp"` (Figure 6.28). The first curve (left) uses the parameter φ and yields a good result. We approximate the curve by a polygon of 200 points.

The second curve (middle) uses the parameter u. The output quality is satisfactory, but only the upper half of the curve is displayed. This is a result of using the arccos function. It returns only positive values.

Using the parameter t in the third curve on the right produces more or less a disastrous result. Again we use 201 points, but their distribution on the curve is really bad. On the right side of the curve you can see clear edges, and you have to increase the number of points considerably in order to smooth them.[6] ♢

Of course, analogous considerations apply to parameterized surfaces as well. Sometimes you may, however, wish to use a certain parametric representation for whatever reasons. Perhaps it is difficult to compute an alternative or you need the parameter lines of that specific representation. The latter was the case with the surface Φ produced by `"conic_section_surface1.cpp"` (Figure 6.29). It is an example of a special surface of conic sections (a surface with a one-parameter set of conic sections on it) investigated in [34].

The parametric curves of the first kind on the surface Φ are the conics. We want to use a special rational parametric representation for them because the parameter lines of the second kind are of interest: They belong to a very special family $\mathcal{F}$ of curves on this surface.

- Each curve of $\mathcal{F}$ lies in a plane of a *fixed pencil of planes* $a(\alpha)$.
- The conics on Φ are *projectively related* by the curves in $\mathcal{F}$.

Furthermore, there exist six straight lines on Φ: the axis a of the pencil of planes and five additonal straight lines $e_0, \ldots, e_4 \in \mathcal{F}$ (parametric lines of the second kind).

In Figure 6.29 two different pictures of Φ are displayed. The image on the left shows the typical drawbacks of the rational parameterization:

- The *distribution of the points* on the conic section is very bad. It produces visible edges (watch the upper right corner of the left image).
- The same is true for the *distribution of the parameter lines* on the surface. They all seem to be attracted by a certain area on the surface and hardly ever occur on the opposite side of Φ.
- The conic sections on the surface are *not closed*. By enlarging the parameter interval, we could more or less get rid of this effect but at the cost of increasing the troubles with the distribution of the points and the parameter lines we have mentioned above.

[6] We have to mention that the rational parametric representation may be very useful in some respect. E.g., the order of c can easily be computed from $\vec{x} = \vec{x}(t)$. It is six if $e \neq r$ and four if $e = r$ (compare [15]).

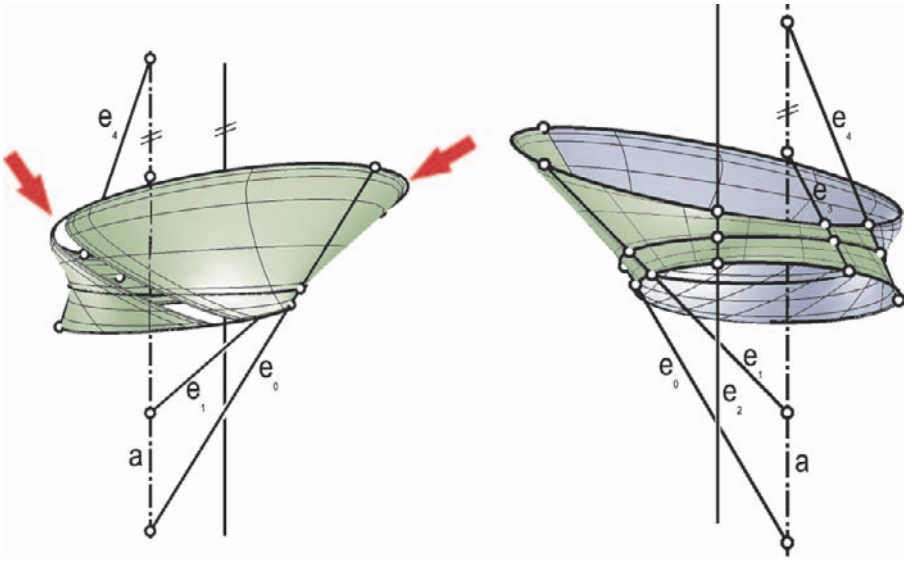

FIGURE 6.29. Bad parameterization of a conic section surface (left, compare `"conic_section_surface2.cpp"`). The surface is split in two and each part is parameterized separately (right, `"conic_section_surface2.cpp"`).

If you now watch the image on the right-hand side, you will notice that all the ugly effects have disappeared. The trick we used is indicated by the two different shading colors of Φ: We split the surface in two parts and parameterized each of them separately! It was not necessary to give up the rational parametric representation. Only a simple parametric transformation was needed.

Example 6.2. Splitting of parameterized curves
We will explain the procedure with a more lucid example from 2D geometry. The basic idea is, of course, the same, and you will be able to use it in spatial geometry as well. Take, for example, the following rational parameterization of a circle c:

$$c \ldots X(t) = \frac{r}{1+t^2}\left(1-t^2, 2t\right)^t.$$

The theoretical parameter interval is $[-\infty, \infty]$. Still, this is not enough to get the curve point

$$U(-r, 0)^t = \lim_{t \to \infty} X(t).$$

Since we can use only a finite parameter interval I, we will always have a gap in a neighborhood of U. For the picture on the left-hand side of Figure 6.30 we chose $r = 5$ and $I = [-12, 12]$. The circle is approximated by a polygon of 150 vertices.

Now we are going to split the circle c in two parts c_1 and c_2 at the highest point $H = X(1)$ and the lowest point $L = X(-1)$. In order to do this, we have to find

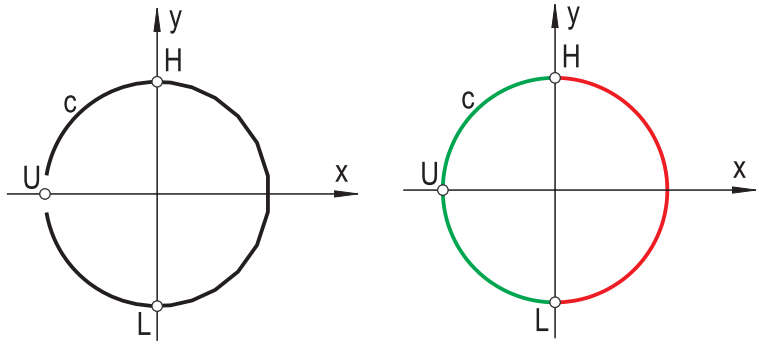

FIGURE 6.30. Rational parametric representation of a circle c (left). The circle is split into two parts that are parameterized and drawn separately (right).

a good rational parametric representation $X_1 = X_1(t)$ of the left part c_2 of c. It is reasonable to claim $X_1(-1) = L$, $X_1(0) = U$, and $X_1(1) = H$. For reasons of symmetry, this parametric representation is

$$c_2 \ldots X_2(t) = r\Big(-\frac{1-t^2}{1+t^2}, \frac{2t}{1+t^2}\Big)^t.$$

The same result can, of course, be obtained by calculating the coefficients a, b, c, and d of a fractional linear map

$$t \mapsto \frac{at+b}{ct+d}$$

that transforms $X(t)$ to $X_2(t)$. This approach is more general. We used it, for example, for the picture on the right-hand side of Figure 6.29 and in Example 3.18. A very detailed discussion of this problem will be given in Example 6.3.

In Figure 6.30 you can see the result of the parametric transformation and splitting. For the circle on the right-hand side (actually two semi circles) we use again $150 = 2 \cdot 75$ points and a rational parametric representation. This time, the result is entirely satisfactory. ♢

Finding good parameterized equations

Frequently, the following problem occurs: Given the parameterized equation of a surface Φ, you want to visualize it on the screen. In principle, this is no problem with Open Geometry. Still, there are certain things to take into account.

Example 6.3. Roman surface
In [39] we find the parametric representation of a surface Φ which the author considers to be very beautiful: Kummer's example of Steiner's Roman surface

([21]); a surface of remarkable properties and high symmetry. The suggested parametric representation for Φ reads

$$\vec{x}(r,\varphi) = \frac{1}{r^2/2+1} \begin{pmatrix} \sqrt{2}(r\cos\varphi + \cos 2\varphi) \\ \sqrt{2}(r\sin\varphi - \sin 2\varphi) \\ r^2 - 1 \end{pmatrix}.$$

We immediately implement this:

```
class MyRomanSurface: public ParamSurface
{
public:
   virtual P3d SurfacePoint( Real r, Real phi )
   {
      Real x, y, z;
      x =  cos( 2 * phi ) + r * cos( phi );
      y = -sin( 2 * phi ) + r * sin( phi );
      z = r * r - 1;
      const Real sqrt2 = sqrt( 2 );
      const Real denom = 0.5 * r * r + 1;
      x *= sqrt2 / denom;
      y *= sqrt2 / denom;
      z /= denom;
      z += 0.5;
      const Real factor = 5;
      return P3d( factor * x, factor * y, factor * z );
   }
};
MyRomanSurface RomanSurface;
```

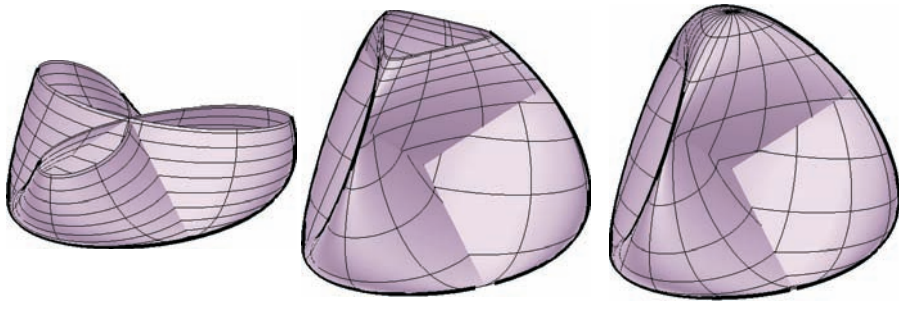

FIGURE 6.31. Parameterizing Kummer's model of the Roman surface.

Now let us have a look at the result (first picture in Figure 6.31).[7] Quite nice, but still, we do not see all of the surface's relevant properties: Φ has no real points at infinity, so we should be able to display a closed surface.

In our first approach we used $(r, \varphi) \in [0, 1] \times [-\pi, \pi)$, but having a look at the geometric meaning of r and φ, we find that $r \in [0, \infty)$ is the maximal parameter range. Thus, we will not be able to see the whole surface without a proper parametric transformation. We can try to increase the parameter range ($r \in [0, 10]$) but the result is not too good (second picture in Figure 6.31). The parameter lines and surface facets are unevenly distributed and we have problems with the contour outline.

Obviously, we have to map the interval $[0, \infty)$ to some finite interval $[0, \alpha)$. For this purpose, a fractional linear parametric transformation seems to be a good idea:

$$r \mapsto f(r) = \frac{ar + b}{cr + d} \qquad ad - bc \neq 0.$$

The conditions $f(0) = 0$ and $f(\alpha) = \infty$ result in $b = 0$ and $c\alpha + d = 0$. After some trial and error we decide on $a = c = 1$ and $\alpha = 2$ as a good choice for our parameters. We insert

```
r = r / ( r - 2 );
```

at the top line of the SurfacePoint(...) function, change the parametric rectangle to $[0, 2] \times [-\pi, \pi]$ and get the result displayed in the third picture of Figure 6.31. That is not too bad. However, if we look at the surface in a bottom view, we find that the parameter lines of one kind look somehow strange. In fact, there exists a theoretical borderline in the middle of the geometrically closed surface Φ, which causes an irritating distribution of the parameter lines.

In this special case (and in many others) it helps to split the surface. That is, we derive two instances RomanSurface1 and RomanSurface2 of the class MyRomanSurface and shade them separately, using $\varphi \in [-\pi, 0]$ and $\varphi \in [0, \pi]$, respectively. After experimenting a little with the number of parameter lines we get the pictures in the third row of Figure 6.32. The bottom view is entirely satisfactory this time.

In the last step we want to emphasize the special properties of Φ. It permits the symmetry operations of a regular tetrahedron. Three double lines are clearly visible on the surface. Their six end points lie on four circles that touch one another. Drawing the circles and the double lines will raise the picture's quality considerably. Some simple spatial trigonometry provides the necessary information, and we can produce the final output (`"roman_surface.cpp"`, Figure 6.32).

[7] We recommend that you follow our thoughts and implement this parameterized surface as well in OPEN GEOMETRY. You can use the templet file `"USER/TEMPLETS/paramsurf.cpp"`.

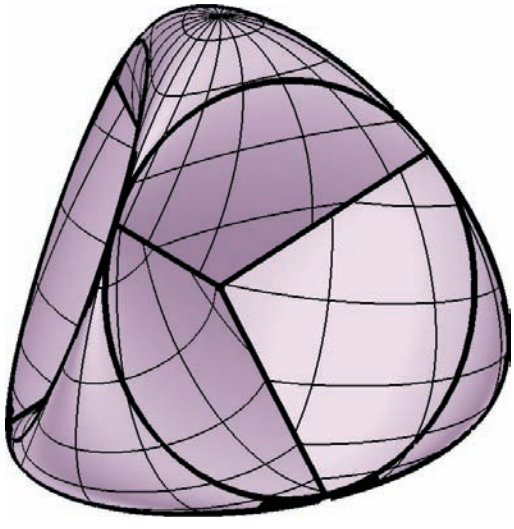

FIGURE 6.32. The final rendering.

♢

A simple example of diverse techniques

The next example is very simple and rather stupid. We have designed it to demonstrate a few OPEN GEOMETRY programming techniques that might be useful to you. These techniques concern fast rotation of a huge number of objects, dynamic memory allocation, and the compiler-safe use of count variables in loops.

Example 6.4. Fast rotation
In `"quickrot2d.cpp"` we apply an ordinary 2D rotation to a huge number of points. Despite the apparent simplicity, we will be able to give a few nontrivial hints, and you will learn more about internal procedures of OPEN GEOMETRY. Since `"quickrot2d.cpp"` is not too long, we display a complete listing:

Listing of `"quickrot2d.cpp"`:

```
#include "opengeom.h"

int NoOfPoints;
P2d *P = NULL;
RandNum Rand;

void Scene::Init( )
{
```

```
    // allocate memory for a huge random number of points
    Rand.Def( 10, 10000 );
    NoOfPoints = (int) Rand.Next( );
    ALLOC_ARRAY( P2d, P, NoOfPoints, "nPoints" );

    // define random points
    Rand.Def( 0, 10 );
    int i;
    for ( i = 0; i < NoOfPoints; i++ )
        P[i]( Rand.Next( ), Rand.Next( ) );

    AllowRestart( );
}
void Scene::Draw( )
{
    for ( int i = 0; i < NoOfPoints; i++ ) // bad style
        P[i].Mark( Magenta, 0.05 );
}
void Scene::Animate( )
{
    int i;
    Rotation2d.Def( Origin, 0.8 );
    for ( i = 0; i < NoOfPoints; i++ )
    {
        // slow rotation
        P[i].Rotate( Origin2d, 0.8 );
        // fast rotation
        //P[i].Rotate( );
    }
}
void Scene::CleanUp( )
{
    // Dont forget to free the dynamically allocated memory!
    if ( P )
        FREE_ARRAY( P2d, P, NoOfPoints, "nPoints" );
}
void Projection::Def( )
{
  if ( VeryFirstTime( ) )
  {
     xyCoordinates( -10, 10, -10, 10 );
  }
}
```

We use three global variables: an integer NoOfPoints, a pointer to an object of type *P2d*, and a random number Rand. In Init( ) we initialize the random number and reserve memory for an array of points. This is necessary because NoOfPoints is not constant. As a consequence, a new random value can be assigned every time you restart the program.

The use of the OPEN GEOMETRY macro ALLOC_ARRAY is very convenient in this context. Do not forget to free the memory at the end of the program by calling FREE_ARRAY. The correct position for this task is CleanUp( ).[8]

After having allocated the memory for the points, we initialize them with random values in Init( ) and mark them in Draw( ). Note the **for**-loop in Draw( ). For our example it is all right, but in general, it is not a good programming style (even though certain books on C++ tell you something different): Depending on your compiler, the scope of i will be either the **for**-loop or the whole Draw( ) part. Problems will arise if you want to use a variable of the same name later in Draw( ).[9] Therefore, we had better use the unambiguous notation

```
int i;
for ( i = 0; i < NoOfPoints; i++ )
   P[i].Mark( Magenta, 0.05 );
```

The animation consists of a rotation with constant velocity about the center of the coordinate system. Again, our solution is not the best. The code line

```
P[i].Rotate( Origin2d, 0.8 );
```

evokes the computation of *the same rotation matrix* for every single point we want to rotate. Thus, our engine has to compute the values of sin(0.8) and cos(0.8) every single time. It would be much better to compute the rotation matrix only once. For such cases, OPEN GEOMETRY provides a global constant called *Global.rotation2d*. It is initialized by

```
Rotation2d.Def( Origin, 0.8 );
```

Thus, we could achieve the same result much faster by toggling comments in Animate( ). Of course, you can speed up 3D rotations in an analogous way.

There exists a second way of speeding up the above code: Replace the angle increment 0.8° by an *integer* value $n \in \{0, \ldots, 360\}$ (e.g., by 1°). This activates

[8] For the dynamic memory allocation of 2D arrays a similar pair of macros exists: ALLOC_2D_ARRAY and FREE_2D_ARRAY.

[9] OPEN GEOMETRY 1.0 has been afflicted by this problem. For the new version, we have corrected all critical code lines.

internal Open Geometry tables of trigonometric functions and prevents the repeated computation of sin(1°) and cos(1°).

If you run the animation twice with an identical number of points but different ways of rotation, you will probably not notice the difference in speed. The reason for this is that the computation of thousands of rotation matrices is still much faster than the marking of thousands of points in Draw( ). It is there that most of the time is lost, despite the fact that Open Geometry handles this task very cleverly.

If you mark one point, Open Geometry does not draw a circle but a little regular polygon. If you mark a second point, this polygon is not recomputed but simply translated to the correct position. Furthermore, the number of polygon vertices depends on its radius: The smaller it is, the fewer vertices are computed. In our example the circles are approximated by twelve-sided polygons. If you want more vertices, you can switch to quality mode by inserting

```
HiQuality( );
```

in Init( ). This will (among other things) double the number of vertices on circles. Note that this is not necessary for the production of high-quality `"*.eps"` files. There, every circle will really be described as a circle. Thus, you don't lose any accuracy. ♢

Accessing object points

Many geometric objects in Open Geometry are derived from the common base class *O2d* or *O3d*. You will find their description on page 460 and page 502, respectively. These classes describe conglomerates of 2D or 3D points and are equipped with methods that permit geometric transformations (translation, rotation, scaling,...) and access to the single member points.

These powerful tools apply to many Open Geometry classes even if you do not find them directly in the corresponding header file. We would like to draw your special attention to such transformation methods as MoveGenerally(...) and MatrixMult(...). Without them, advanced animations as in `"oloid.cpp"` would be much harder.

Sometimes, it is very useful to directly access the member points of a geometric object. If the object is derived from *O2d* or *O3d*, this is possible. Both classes have the common ancestor *O23d* with the *public* member variables

```
int nPoints;
P3d *pPoints;
```

The variable nPoints gives the number of stored object points, pPoints is a pointer to them. You can address a point by means of the [] operator, i.e., you may to write

P3d P = nPoints[i];

where i ranges in [1, nPoints]. Note that you are even allowed to change the value of nPoints. Please, don't do that if you are not really aware of the consequences! We even recommend addressing nPoints indirectly via Size( ), because we intend to declare nPoints as **private** in some future version. Under no circumstances should you increase nPoints. This would almost immediately result in a compilation error or program crash. In order to avoid these troubles, you should use the *O23d* method SetSize(...). It allows only the decreasing of nPoints. This is permissible and sometimes even sensible.

Numerical errors

Open Geometry provides a few additions to the standard catalogue of C's mathematical functions (compare page 557). Some are entirely new; others are just alternatives to the standard functions. Usually, Open Geometry versions are equipped with security checks and catch undefined cases. Using them might prevent your system from crashing and avoid hard-to-find errors. We have a stunning example of this:

Example 6.5. Buggy kardan
The file `"buggy_kardan.cpp"` is an almost identical copy of `"kardan.cpp"`. Starting it, you will be presented a wire frame model of a kardan joint. Restart the program immediately by pressing <`Ctrl + Shift + R`>. This switches to a solid model of a teaching device: With the help of a little crank, one can drive the kardan joint. Now, start the animation <`Ctrl + F`> and wait a little. The angle between the two axes of rotation will be increased now and then until, suddenly, something strange happens (somewhere around frame number 240).

Can you find the bug without comparing our sample file to `"kardan.cpp"`? There is only one different code line:

Omega = Deg(acos(v ∗ v0));

If you replace this line by

Omega = Deg(ArcCos(v ∗ v0));

everything will be all right. Here v and v0 are 3D vectors. Directly before computing the inverse cosine of their dot product, they are normalized. Still, due to numerical errors, the inequality $-1 \leq \mathsf{v} * \mathsf{v0} \leq 1$ is not guaranteed! You can test it by inserting

```
if fabs( ( v ∗ v0 > 1 ) )
    SafeExit( );
```

directly before the line in question. Open Geometry's ArcCos(...) function takes care of bad input data; C's acos(...) does not. As a result, the middle cross of the model will be hurled to some place in vast 3-space.

Finding this error was really hard work. One would expect that the dot product of normalized vectors is within range, but this is not the case. Therefore, it is really advisable to use frequent security checks. Compared with the computation of the inverse cosine, the time for the additional check can be neglected. ♢

This page intentionally left blank

7

How to Expand Open Geometry

This chapter is written for advanced Open Geometry users with some programming experience. It is not written for C++ beginners! We will deal with items such as:

- multiple scene programs,
- line scope of variables,
- workspace,
- dynamic memory,
- privacy of classes,
- customizing Open Geometry to your special preferences.

7.1 Multiple Scenes

In Open Geometry 1.0 we could not compile two or more programs at once. We had to put the old **#include** line in `"try.cpp"` in comments before including the next example. The reason for this is that we are obliged to implement the virtual functions Init( ), Draw( ), Animate( ), CleanUp( ), and Projection::Def( ), but we are not allowed to implement them more than once. Essentially, this is still the case in Open Geometry 2.0. However, we introduced a new possibility that allows the compilation of several programs at once.

Example 7.1. Multiple scenes
The production of a multiple scene file needs two steps. First you must prepare every single program. Make sure that they all compile and yield the desired result. Then you have to write a *nonstandard* OPEN GEOMETRY file. The following listing shows a sample where seven applications are compiled at once:

Listing from "multiple_scenes.cpp":

```
#include "opengeom.h"

int get_scene_number( )
{
   static int i = 0;
   i++;
   if ( i > 7 )
      i = 1;
   return i;
}

#include "scene1.h"
#include "X/SCRIPT/running_3d_wave.cpp"

#include "scene2.h"
#include "X/3D/develop.cpp"

#include "scene3.h"
#define F FScene3
#include "X/SCRIPT/f_deriv.cpp"
#undef F

#include "scene4.h"
#define N NScene4
#include "X/SCRIPT/gg.cpp"
#undef N

#include "scene5.h"
#define Surf SurfScene5
#include "HANDBOOK/BSPLINE/nubs_surf.cpp"
#undef Surf

#include "scene7.h"
#include "X/3D/IMPORT/teapot.cpp"

#include "scene6.h"
#include "BOOK/paramcurve2d.cpp"

#include "multiple_scenes.h"
```

We call this file "nonstandard" because Init( ), Draw( ), Animate( ), CleanUp( ), and Projection::Def( ) are missing. In "multiple_scenes.cpp" we do three things:

1. We include "multiple_scenes.h" at the bottom of "multiple_scenes.cpp".
2. We implement a function get_scene_number(...) that returns an integer value $1 \leq i \leq 7$. The maximum number of scenes to be displayed is 30. In order to change this number, you have to edit "multiple_scenes.h". We will soon explain this more precisely.
3. We include our different scene files as displayed in the listing. Here, it is important to *redefine every global variable name that occurs in different included scenes!* In our example these are the variables F, N, and Surf. Don't worry if you forget to do this; the compiler will tell you. Note that we deliberately use 2D and 3D scenes. This has no negative effect on the compilation process.

All three points have an important meaning. Our get_scene_number(...) function is not very sophisticated. The return value is increased with every restart of the program (menu item "Program→Restart Program" or shortcut <Ctrl+Shift+r>). Thus, you will see scene 1, scene 2,...,scene 7 and then start again with scene 1. One could as well write a function that reacts to keystrokes or implement a pulldown menu and choose the scenes via a mouse click.

We have to avoid ambiguous variable names. This is done by redefining them before the respective **#include** statements. This solution is not very elegant but has a big advantage: *We do not need to change previously written* OPEN GEOMETRY *programs and can preserve the backward compatibility of your* OPEN GEOMETRY *version.*

The first point, inclusion of "multiple_scenes.h", is of interest, too. We list the source code to give you a clue as to what is happening:

Listing from "H/multiple_scenes.h":

```
#undef Scene
#undef Projection

Projection *CurCamera;
Scene *CurScene;

int CurSceneNumber;

void Scene::Init( )
{
  CurSceneNumber = get_scene_number( );
```

```
   if ( VeryFirstTime( ) )
   {
      switch ( CurSceneNumber )
      {
#ifdef __SCENE1__
         case 1: CurScene = new Scene1; break;
#endif

...

#ifdef __SCENE30__
         case 30: CurScene = new Scene30; break;
#endif
      default:
         SafeExit( "Wrong number of scene" );
      }
      TheScene = Global.application = *CurScene;
      CurScene->Init( );
   }
   AllowRestart( );
}
void Scene::Draw( )
{
   CurScene->Draw( );
}
void Scene::Animate( )
{
   CurScene->Animate( );
}
void Scene::CleanUp( )
{
   CurScene->CleanUp( );
}
void Projection::Def( )
{
   if ( VeryFirstTime( ) )
   {
      Projection *cam = NULL;
      switch ( CurSceneNumber )
      {
#ifdef __SCENE1__
         case 1: cam = new Camera1; break;
#endif

...

#ifdef __SCENE30__
         case 30: cam = new Camera30; break;
#endif
```

```
            default: SafeExit( "bad number of scene" );
        }
        CurCamera = Global.camera = cam;
        CurCamera->Init( );
    }
    CurCamera->Def( );
}
```

Probably, you have already wondered what has happened to Init( ), Draw( ), Animate( ) and CleanUp( ). They have moved to the header file. There, the pointer CurScene sends them to the respective function of the current scene. In order to allow this, some major changes to the inner structure of OPEN GEOMETRY were necessary. However, you should not even notice them unless you view `"multiple_scenes.h"`. ♢

7.2 Global and Static Variables

In this section we will talk about the scope of a variable and the impact of a global variable.

Your OPEN GEOMETRY program is, in general, relatively short, say two or three pages on the screen. Everything that is declared outside the four member functions of *Scene* — Init( ), Draw( ), Animate( ), CleanUp( ) — and Projection::Def( ) is "global" for the rest of the program. If a variable is declared before all these functions, it can be defined and changed at any position. In the same way, a function can be used everywhere, as long as its prototype is listed on top of the program (the implementation can be anywhere else).

If the declaration of the global variable or the function is not protected by the keyword **static**, other modules have direct access to this variable by means of the keyword **extern**.[1]

However, as a general rule, please obey the following: *Always initialize a global (nonconstant) variable in the* Init( ) *part.* This is the only safe way to make a program restartable. Thus, you should *not initialize global variables by means of constructors*, although in many cases OPEN GEOMETRY would allow you to do so. An example will show you the reason for this step by step: The program

[1]The library of OPEN GEOMETRY, of course, uses this feature quite often; it is a convenient way of "communication" between the modules. Nevertheless, this library does not use "ordinary" variables: Instead, a single instance named *Global* of a large structure called *GlobalVariables* was introduced. The structure — and with it all global data — can be initialized, reinitialized, and deleted on demand (e.g., for a restart of the program). Besides the variable *Global*, there is only one additional global Boolean variable, named *GlobVarInitialized*, which is initialized with false when the program is started.

```
#include "opengeom.h"
#include "defaults3d.h"

Box B( Green, 5, 5, 5 ); // B is initialized via constructor

void Scene::Init( )
{
}
void Scene::Draw( )
{
   B.Shade( );
}
```

displays a cube correctly. Since the cube is never altered, the program even allows a restart:

```
#include "opengeom.h"
#include "defaults3d.h"

Box B( Green, 5, 5, 5 ); // B is initialized via constructor

void Scene::Init( )
{
   AllowRestart( );
}
void Scene::Draw( )
{
   B.Shade( );
}
```

This is not trivial: The variable B contains dynamic memory that is deleted when the destructor is called. When a program is restarted, however, its destructor is *not* called automatically (you would have to call the destructor explicitly in CleanUp( )). Note that a restart will *not* call the constructor again!

Next, the program

```
Box B( Green, 5, 5, 5 ); // B is initialized via constructor

void Scene::Init( )
{
   B.Translate( 1, 0, 0 );
   AllowRestart( );
}
void Scene::Draw( )
{
   B.Shade( );
```

```
}
```

is no longer restartable: Every restart calls Init( ). There, our box is translated. With every restart, the cube will appear at another starting position.

Here is finally the restartable version: The box is initialized (defined) in Init( ):

```
Box B; // B is not initialized!

void Scene::Init( )
{
    B.Def( Green, 5, 5, 5 );
    B.Translate( 1, 0, 0 );
    AllowRestart( );
}
void Scene::Draw( )
{
    B.Shade( );
}
```

For some reasons, e.g., for a multiple scene application, it may be necessary to count the number of restarts. With the above, this can be done with the following code:

```
int NoOfRestarts = -1;

void Scene::Init( )
{
   NoOfRestarts++;
   AllowRestart( );
}
void Scene::Draw( )
{
}
```

If we do not need the variable NoOfRestarts globally, we can introduce a "pseudo global variable" by means of the keyword **static** as follows:

```
void Scene::Init( )
{
   static int noOfRestarts = -1;
   noOfRestarts++;
   AllowRestart( );
}
void Scene::Draw( )
{
```

```
}
```

In this case there is no way to have access to noOfRestarts from outside. If we want to forbid access from other modules to our personal global variables, we can use the **static** keyword in its second meaning:

```
static int NoOfRestarts = −1;

void Scene::Init( )
{
   NoOfRestarts++;
   AllowRestart( );
}
void Scene::Draw( )
{
}

// in some other module...
int no_of_restarts( )
{
   extern int NoOfRestarts;
     // only works, if NoOfRestarts is not static
   return NoOfRestarts;
}
```

7.3 The Initializing File "og.ini"

Open Geometry is a programming system that needs a compiler. Once an application is compiled, it will run with clearly defined parameters. For some reasons, however, it is useful to be able to change parameters "from outside," for example, by means of an initializing file.

Since "learning by doing" is a good method, we give a sample listing of the initializing file `"og.ini"`:

```
begin Open Geometry parameters

please change only parameters

  AllowRestart no AllowRestart( )

  HardwareBug no

  NameOfOutputDirectory ./

  PathTo_BMP_Directory ./

  PathTo_DATA_Directory ./
```

```
   ChangePovRayLineWidth 1
   ChangePScriptLineWidth 1.8
   Color Yellow low_rgb 0.45 0.25 0.0 high_rgb 1 1 0.5
end Open Geometry parameters
```

In principle, OPEN GEOMETRY reads this file before the scene's member function Init( ) is called. When you write

```
AllowRestart yes
```

this has the same effect as using the routine AllowRestart( ) in the Init( ) function. Note that the predefinition can be overwritten by AllowRestart(false); i.e., the code in the application has priority.

Analogously, you can predefine line widths of the EPS output or the POV-Ray output when you do not like the predefined values at all, but you are not willing to write the corresponding commands into each application file.[2]

The line

```
HardwareBug yes
```

can sometimes help to overcome some hardware bugs: Many video cards support OPENGL via hardware. Some of them, though, have minor bugs, mostly in connection with "thick lines," obviously for speed reasons:

- Some cards cannot display "good-looking" thick lines. The line should actually be a slim rectangle. However, it appears as a parallelogram either horizontally or vertically, which is disturbing when a thick line rotates during an animation.
- This error is even worse: When a thick line has to be clipped with the screen on different sides, it is sometimes not drawn at all.

Another useful thing is this: You can change the predefined color palettes according to your taste. Say, you do not like the predefined *Yellow* palette, then just try out your own palette; by means of

```
Color Yellow low_rgb 0.45 0.25 0.0 high_rgb 1 1 0.5
```

The RGB values have to be between zero and one. By the way, this is a quick method to create a good-looking palette: No compilation is necessary!

[2]You could change the code in `"h.cpp"`, where these variables are initialized, but remember: These changes are undone with every update!

When you are satisfied, you can initialize the palette with the command *CreateNewPalette*(...) (compare page 569).

Finally, a last and important feature: You can change the standard input and output directory. Say, your disk partition is full and you want to export a huge file. Then redirect the output to another directory. Do not forget the slash at the end. The default directory is

```
NameOfOutputDirectory ./
```

You can load your bitmap files from another directory or change the `"DATA/"` input by writing something like

```
PathTo_BMP_Directory c:/temp/
```

In this directory, however, the files have to be stored in the subdirectory `"BMP/"` or `"DATA/"`. This is due to the fact that most Open Geometry applications include files from subdirectories with these names.[3]

The latter features can also be used for the following: Your Open Geometry "home directory" is somewhere on your disk (say in `"D:/OPENGEOM/"`), but you have precompiled demoversions on your desktop. Then put an `"og.ini"` on the desktop where the corresponding directories point to your home directory. Thus, you do not have to copy all your data to the desktop. The corresponding `"og.ini"` lines may then look like this:

```
NameOfOutputDirectory c:/temp/
PathTo_BMP_Directory d:/opengeom/
PathTo_DATA_Directory d:/opengeom/
```

7.4 How to Make Essential Changes

In this section we will show you how to wire in essential changes and class implementations and still stay compatible with future versions of Open Geometry. This is especially important for people who are willing to contribute to the development of Open Geometry and/or are working intensively with Open Geometry.

First of all, we list again the general rules:

- Especially if you intend to write multi scene applications, write programs that do not leave memory leaks when you quit them. Therefore, when you:

[3] Of course, you will get the same effect by passing the complete path as parameter.

  - write classes of your own that contain dynamically allocated memory, you should definitely equip them with a constructor and a destructor.
  - work with global dynamic arrays, always check whether they are to be freed in the CleanUp( ) part.

- Furthermore, you know already from the introductory section that you should:
  - definitely not overwrite the distributed files `"d.cpp"`, `"e.cpp"`, `"f.cpp"`,... of the C directory.
  - also better not put your "personal header files" into the directory `"H/"`. Preferably, put them into the directory `"USER/H/"`.

- A last thing you are asked for (and what at the first sight sounds unusual to an advanced programmer): Do not add any new files to the workspace! Every new version will overwrite the workspace. In the following, we will show you that this is not necessary.

Here is an artificial example of an essential change.

Say, you would like to use the class *Function* that is declared in the distributed header File `"function2d.h"`. However, you need an additional member function that determines extreme points like minima and maxima (and also the saddle points; Figure 7.1). How can you do this with a minimum of work?

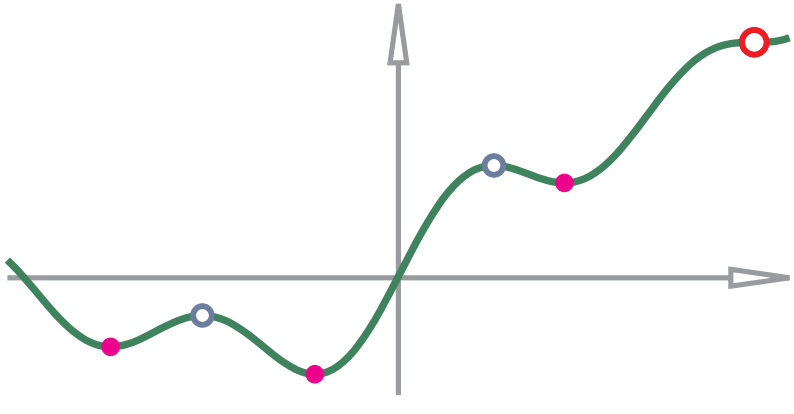

FIGURE 7.1. A function graph with minima, maxima, and a saddle point (output of both `"introduce_new_class.cpp"` and `"test_funct_min_max.cpp"`).

- In the first step you develop a new application `"introduce_new_class.cpp"`

as usual. There you implement a class MyFunction[4] that is derived from the predefined class *Function*. This class inherits "everything" from the parent class, and you can additionally implement a new member function.

If you take a look at the listing of `"introduce_new_class.cpp"`, you can see what the new class MyFunction can do: It allows the calculation of minima, maxima and saddle points of an arbitrary function graph. The implementation of the member function GetMinMax(...) makes use of the member functions FirstDerivative(...) and SecondDerivative(...) of the parent class. It passes parameters that are of type *PathCurve2d*. The size of the respective path curve is the number of minima/maxima/saddle points, and its points are the corresponding extreme points of the function graph. After browsing over the code, you will probably agree that the new feature is worth being included into the whole system.

Listing of `"introduce_new_class.cpp"`:

```
#include "opengeom.h"
#include "defaults2d.h"

#include "function2d.h"
class MyFunction: public Function
{
public:
   void GetMinMax( Real x1, Real x2,
                   PathCurve2d &min,
                   PathCurve2d &max,
                   PathCurve2d &saddle,
                   int n = 800, Real eps = 1e-3 );
};

MyFunction F;

Real f( Real x )
{
   return x/2 + sin( x ) + sin( x / 2 ) + Sqr( x - 1 ) / 30.65;
}
void Scene::Init( )
{
   F.Def( f );
}
```

[4]Note that the name of the class was not recognized as an Open Geometry class by our beautifying program. Thus, it is not written in italics, and it was not automatically put into the index of the book!

```
void Scene::Draw( )
{
   ShowAxes2d( DarkGray, −10, 10, −3, 7 );
   F.Draw( Green, −10, 10, 150, MEDIUM );
   PathCurve2d min, max, saddle;
   F.GetMinMax( −10, 10, min, max, saddle, 1000, 1e−2 );
   min.MarkPoints( Magenta, 0.15 );
   max.MarkPoints( DarkBlue, 0.2, 0.1 );
   saddle.MarkPoints( Red, 0.25, 0.15 );
}

// implementation of the member function of MyClass

void MyFunction::GetMinMax( Real x1, Real x2,
                            PathCurve2d &min,
                            PathCurve2d &max,
                            PathCurve2d &saddle,
                            int n, Real eps )
{
   Real dx = ( x2 − x1 ) / ( n − 1 ) + 1e−12, left_x, right_x;
   Real left_y, right_y;
   for ( left_x = x1; left_x < x2; left_x += dx )
   {
      right_x = left_x + dx;
      left_y = FirstDerivative( left_x );
      right_y = FirstDerivative( right_x );
      if ( left_y * right_y < 0 )
      {
         P2d p( left_x, left_y ), q( right_x, right_y );
         P2d s = StrL2d( p, q ) * Xaxis2d;
         s.y = (*this)( s.x );
         Real f2 = SecondDerivative( s.x );
         if ( fabs( f2 ) < eps )
            saddle.AddPoint( s );
         else if ( f2 < 0 )
            max.AddPoint( s );
         else
            min.AddPoint( s );
      }
   }
}
```

- The "wire-in phase" is now the second step. It is worth following the recipe step by step:

  1. Create a new header file, e.g., `"my_function.h"`, preferably in the `"USER/H/"` directory. There, you put the declaration of MyFunction (see listing below).
  2. Implement the class in the already existing file `"add_code.cpp"` in the directory `"USER/"` (see listing of `"add_code.cpp"`). This file is automatically included in the workspace. Do not add other files to the workspace!
  3. Write your application, say `"test_funct_min_max.cpp"`, where you can use the new class exactly the same way as you are used to (see listing of `"test_funct_min_max.cpp"`), and enjoy again the output (Figure 7.1).
  4. When you think that you have done a good job and you have implemented a useful feature, contact us and send us the code together with a short description. We will integrate your work in a future update and distribute it to other users. For more information, please refer to the Open Geometry home page

     `http://www.uni-ak.ac.at/opengeom/`

     Remember: Open Geometry should remain *open* to all those who love geometry.

Listing of `"USER/H/my_function.h"`:

```
#ifndef __MYFUNCTION_H__

#include "function2d.h"
class MyFunction: public Function
{
public:
   void GetMinMax( Real x1, Real x2,
                   PathCurve2d &min,
                   PathCurve2d &max,
                   PathCurve2d &saddle,
                   int n = 300,
                   Real eps = 1e-3 );
};

#define __MYFUNCTION_H__
#endif
```

Listing of "USER/add_code.cpp":

```
#include "USER/H/my_function.h"

void MyFunction::GetMinMax( Real x1, Real x2,
                            PathCurve2d &min,
                            PathCurve2d &max,
                            PathCurve2d &saddle,
                            int n, Real eps )
{
   // see listing of "introduce_new_class.cpp"
}
```

Listing of "test_funct_min_max.cpp":

```
#include "opengeom.h"
#include "defaults2d.h"

#include "USER/H/my_function.h"

MyFunction F;

Real f( Real x )
{
   return x/2 + sin( x) + sin( x / 2 ) + Sqr( x - 1 ) / 30.65;
}
void Scene::Init( )
{
   F.Def( f );
}

void Scene::Draw( )
{
   ShowAxes2d( DarkGray, -10, 10, -3, 7 );
   F.Draw( Green, -10, 10, 150, MEDIUM );
   PathCurve2d min, max, saddle;
   F.GetMinMax( -10, 10, min, max, saddle, 1000, 1e-2 );
   min.MarkPoints( Magenta, 0.15 );
   max.MarkPoints( DarkBlue, 0.2, 0.1 );
   saddle.MarkPoints( Red, 0.25, 0.15 );
}
```

7.5 Dynamic Memory

In several examples we have already talked about dynamic memory allocation. Beginners tend to avoid this delicate matter, but the longer you work with Open Geometry, the more you will feel the need to use it. Many of Open Geometry's classes would not be possible without dynamic memory allocation. Usually, the user need not even be aware of it because allocation and deletion are handled automatically by the "intelligent" classes.

In the following we will describe how to allocate and free memory dynamically *with the help of* Open Geometry. You may need it for one or the other of your programs, but more likely if you want to create classes of your own. The topic has already been dealt with in [14], but we consider it of fundamental importance. Therefore, we will repeat the main ideas.

Allocating and deleting memory

In C++, memory should be allocated by means of the **new** operator, and it should be freed by the **delete** operator, as in the following example:

```
int *x;
x = new int;
*x = 100;
delete x;
```

In order to avoid the asterisk $*$ with every call of the above variable x one can use the following syntax:

```
int *x;
x = new int;
int &y = *x;
y = 100;
delete x;
```

Note that the value of y does not make sense after x has been deleted.

Dynamic arrays (the size of which are calculated at runtime) are allocated as follows:

```
int *x;
int n = 100;
x = new int [n];
x[23] = 100;
delete [ ] x;
```

Note the order of the brackets [ ].

The allocating and deleting of memory are some of the most sensitive processes in computer programming. Thus, OPEN GEOMETRY provides macros for the allocation of dynamic arrays that are "safe." These macros have been tested out thoroughly, and we recommend that you use them — and only them — unless you know exactly what you are doing. An example of macros of that kind is displayed in the following listing:

```
int *x = NULL;
int n = 100;
ALLOC_ARRAY( int, x, n, "x" );
x[23] = i;
// etc.
FREE_ARRAY( int, x, n, "x" );
```

It is implemented in `"alloc_mem.h"`:

Listing from `"H/alloc_mem.h"`:

```
#define ALLOC_ARRAY( typ, array, size, str )\
{\
   if ( ( size ) == 0 )\
   {\
      SafeExit( StrCat("0 bytes for new ", str ) );\
   }\
   else\
   {\
      if ( array != NULL )
         Write( StrCat("set pointer to zero:", str ) );\
      array = new typ [( size )];\
   }\
}
#define FREE_ARRAY( typ, array, size, str )\
{\
   if ( array == NULL )\
   {\
      Write( StrCat("free NULL pointer:", str ) );\
   }
   else {\
      delete [ ] array;\
      array = NULL;\
   }\
}
```

"Intelligent arrays"

As a very useful application, we now show how to develop C++ classes that handle arrays perfectly. The idea is that the array itself is hidden as a member of the class (structure). Inline operators allow the elements of the array to be read from and written to.

The goal is to write code like the following:

```
IntArray x( 10000 ), y;

RandNum rnd( -100, 100 );
    // initialize a random number in [-100,100]

int i;
for ( i = 0; i < x.size; i++ )
    x[i] = rnd( );

y = x; // The = operator has to be overwritten
y.Sort( );
Print( y.Average( ) );
    // by the way: in this case, the average value should converge to 0

// etc.
```

This is the corresponding structure. The constant DUMMY is used in several other implementations.

Listing from `"H/intarray.h"`:

```
#ifndef __INT_ARRAY__

struct IntArray
{
   int size;
   int *val;
   IntArray( ) { size = 0; val = NULL; }
   IntArray( int n, int init_val = DUMMY  )
   {
      val = NULL; size = 0;
      Def( n, init_val );
   }
   void Def( int n, int init_val = DUMMY );
   int & operator [ ] ( int i ) const
   {
      if ( i > size )
         Write("IntlArray:idx ", i - size , DUMMY, "too high" );
```

```
        return val[i];
    }
    void operator = ( const IntArray &other )
    {
        other.Copy( *this );
    }
    void Copy( IntArray &f ) const;
    void Sort( int i1 = 0, int i2 = -1, int sgn = 1 );
    int Max( ) const;
    int Min( ) const;
    Real Average( ) const;

    void Delete( )
    {
        if ( val )
            FREE_ARRAY( int, val, size + 1, "cont");
    }
    virtual ~IntArray( )
    {
        Delete( );
    }
};

#define __INT_ARRAY__
#endif // __INT_ARRAY__
```

The meaning of the member functions Copy, Sort, Max, Min, and Average is obvious. An analogous structure is *RealArray* (defined in `"realarray.h"`).

Higher-dimensional arrays

Sometimes we have to allocate *higher-dimensional arrays* dynamically. In Open Geometry this is necessary, for example, for tensor product surfaces that are defined via a 2D array of control points. However, in order to present the basic ideas, let us talk about a more abstract case. We want to allocate the two-dimensional array **int** x[n][m]. The array has n m elements of type **int**. The expression x[i][j] is turned into *(x[i] + j) by the compiler. Thus, the computer needs information about the n pointers x[i]. Therefore, the allocation of the array has to be done in three steps:

1. Allocate the n pointers.
2. Allocate space for the entire array.
3. Initialize the rest of the pointers.

The array is freed in two steps:

1. Free the space for the entire array.
2. Free the space for the n pointers.

It is fairly complicated to rewrite such code for every new allocation, and there is always the danger of memory errors. This problem cannot be solved by a function, because we have to distinguish between different types of variables if the pointers are to be cast correctly. Once again, a C macro comes in handy:

```
int **x = NULL;
int n1 = 1000, n2 = 500;
ALLOC_2D_ARRAY( int, x, n1, n2, "x" );
for ( int i = 0; i < n1; i++ )
   for ( int j = 0; j < n2; j++ )
      x[i][j] = i + j;
FREE_2D_ARRAY( int, x, n1, n2, "x" );
```

The Open Geometry implementation of the macros is as follows:

Listing from `"H/alloc_mem.h"`:

```
#define ALLOC_2D_ARRAY( typ, array, size1, size2, str )\
{\
   ALLOC_ARRAY( typ *, array, size1, str );\
   array[0] = NULL;\
   ALLOC_ARRAY( typ, array[0], ( size1 ) * ( size2 ), str );\
   for ( int i = 1; i < size1; i++ )\
      array[i] = array[ i - 1] + size2;\
}
#define FREE_2D_ARRAY( typ, array, size1, size2, str )\
{\
   if ( array == NULL )\
      Write( StrCat("free NULL pointer *:", str ) );\
   else\
   {\
      FREE_ARRAY( typ, array[0], ( size1 ) * ( size2 ), str );\
      FREE_ARRAY( typ *, array, size1, str );\
   }\
}
```

7.6 An Example of a Larger Project

In this section several elliptic compasses are studied. For an overview, please start the multiple-scene application

```
"elliptic_compasses.exe"
```

and have a look at the different mechanisms. At the end of the section "compasses" for the drawing of parabolas and hyperbolas are introduced.

Introduction[5]

In a workshop on kinematics held at Vienna's University of Applied Arts a student presented an interesting mechanism of an elliptic compass (Figure 7.5). This chapter is based on our attempts to produce an animation of this elliptic compass (and others as well) with Open Geometry 2.0.

Some of the supplied programs are to be found on the accompanying CD-ROM in the directories `"X/KH/CONIC_COMPASS/"` and `"X/KH/KINE/"`. Others can be downloaded from our home page

```
http://www.uni-ak.ac.at/opengeom/
```

Instead of constructing a number of ellipse points with ruler and compass or draw the ellipse with the help of a computer, we can use a technical stencil. Another, more flexible, possibility is the use of an elliptic compass.

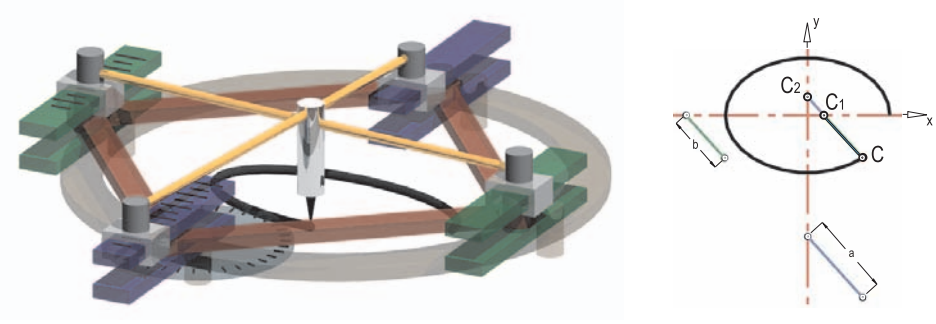

FIGURE 7.2. Elliptic compass of Hoecken.

A very simple way to get an elliptic path curve is to fix a thread on two points and move a pencil along the stretched thread (Figure 7.3). This is in line with the proposition that all points with equal sum of distances to two fixed points lie on an ellipse. Gardeners exchange the pencil with a spade to get an elliptic flowerbed.

[5]In this section, G. Karlhuber summarizes his Open Geometry project on elliptic compasses.

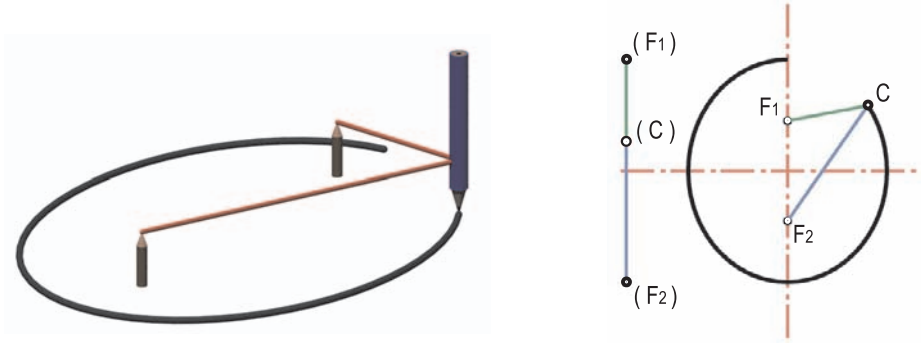

FIGURE 7.3. The gardener method; $\overline{CF}_1 + \overline{CF}_2 = \text{const}$.

The following, more sophisticated, mechanisms of elliptic compasses are based on planar kinematics and are sorted by theory. At the end we will also discuss inversion and special trochoid motions for constructing conic compasses.

As already mentioned, the particular programs are stored in `"X/KH/KINE/"` and `"X/KH/CONIC_COMPASS/"`. The header files are included from `"X/KH/H/"` and CAD3D objects are imported from `"X/KH/DATA/"`. In `"elliptic_compass.h"` you will find the common parts of all the programs:

Listing of `"X/KH/H/elliptic_compass.h"`:

```
void Projection::Def( )
{
   if ( VeryFirstTime( ) )
   {
      DefaultCamera( 30, 40, 50 );
   }
}

#ifndef __ELLIPTIC_COMPASS__

class EC
{
public:
   void Init( );

   // global hotkeys for interaction
   void explain_keyboard_fix( );
   void draw_paper_sheet( );

   // drawing an ellipse
   void show_ellipse( );
```

```
    P3d Pen;
    int pen, t, tStop;
    RegPyramid penPyramid;
    RegPrism penPrism;
    Real a, b, da, db; // lengths of the semiaxis
    PathCurve3d PathX; // path curve of the ellipse
    Boolean Drawing, Visible, Visible2;
};

void EC::draw_paper_sheet( )
{
    Rect3d paper;
    paper.Def( PureWhite, 20 * sqrt( 2 ), 20 );
    paper.Translate( -14, -10, 0.0 );
    SetOpacity( 0.5 );
    paper.Shade( );
    SetOpacity( 1.0 );
}

void EC::explain_keyboard_fix( )
{
    // global hotkeys
    PrintString( DarkBlue, -20, 19, "a =%.2f", a );
    PrintString( DarkGreen, -20, 17, "b =%.2f", b );
    PrintString( Black, 8.5, 19, "press 'Ctrl'+'f',then" );
    PrintString( Black, 8.5, 17, "draw with 'd'" );
    PrintString( Black, 8.5, 15, "try also 'c'and 's'" );
}

void EC::show_ellipse( )
{
    // drawing of the ellipse
    if ( Drawing )
    {
        PathX.Draw( MEDIUM );
    }
}

void EC::Init( )
{
    Drawing = false;
    Visible = true;
    Visible2 = false;

    PathX.Def( Black, 30000 );
}

#define __ELLIPTIC_COMPASS__
#endif  __ELLIPTIC_COMPASS__
```

Two-point guidance

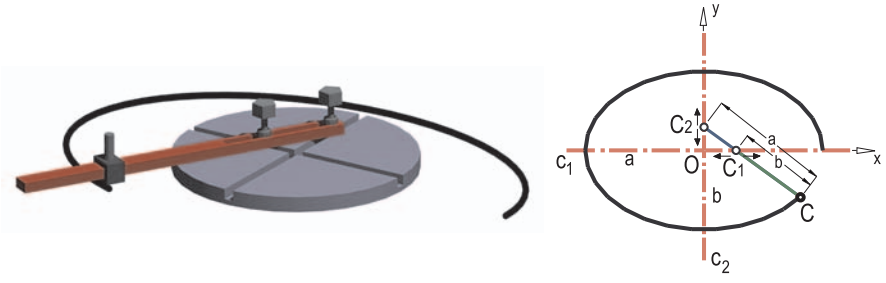

FIGURE 7.4. Mechanism using the two-point guidance.

Two linked points C_1, C_2 glide along two nonparallel straight lines. According to the theorem of De La Hire, each point on the line $\overline{C_1C_2}$ moves on an ellipse (special cases are circles and straight lines). The first ellipsographs were variations of the two-point guidance with perpendicular lines c_1 and c_2. Let $O = c_1 \cap c_2$ be the origin of a 2-dimensional Cartesian coordinate system and C a point of $\overline{C_1C_2}$ (Figure 7.4).

The coordinates of C are

$$x = a \cdot \cos\varphi, \quad y = b \cdot \sin\varphi.$$

This can be transformed to the well-known equation

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

of an ellipse. The ellipsograph shown in Figure 7.4 is the simplest representative of this sort. All programs of this mechanisms have a common mask:

Listing of `"X/KH/CONIC_COMPASS/mask_elliptic_compass.cpp"`:

```
#include "opengeom.h"

// do not change this include file
#include "X/Kh/H/elliptic_compass.h"

class EllipticCompass : public EC
{

public:
   void Init( );
   void ChangeCompass( ); // animation
   void Draw( );
```

```
private:
   // local hotkeys needed in the .h file
   void explain_keyboard( );

   // kinematics of this compass
   void get_theory( );
   // draw kinematic background
   void show_theory( );

   void draw_compass( );

  // add your variables and methods here
};


EllipticCompass EcDummy;

void Scene::Init( )
{
   EcDummy.Init( );
}

void Scene::Draw( )
{
   EcDummy.Draw( );
}

void Scene::CleanUp( )
{
}

void Scene::Animate( )
{
   EcDummy.ChangeCompass( );
}

void EllipticCompass::get_theory( )
{
   // add the kinematic backgrounds here
}

void EllipticCompass::show_theory( )
{
   // do the drawing of kinematic backgrounds here
}

void EllipticCompass::draw_compass( )
{
   // do the drawing of your elliptic compass here
```

```
}

void EllipticCompass::explain_keyboard( )
{
   // write your local hotkeys here
}

void EllipticCompass::Draw( )
{
   if ( Drawing ) show_ellipse( );

   // declared in the particular programs
   if ( Visible2 ) show_theory( );
   if ( Visible ) draw_compass( );

   draw_paper_sheet( );

   explain_keyboard_fix( );
   // declared in the particular programs
   explain_keyboard( );
}

void EllipticCompass::ChangeCompass( )
{
   // add your animation here

   if ( da != 0 || db != 0 )
   {
      PathX.Def( Black, 30000 );
   }

}

void EllipticCompass::Init( )
{
   // initialize everything here

   Drawing = true;
   Visible = true;
   Visible2 = false;

   PathX.Def( Black, 30000 ); // path curve of the ellipse

   // lengths of the semiaxis
   a = 6;
   b = 2;
   da = 0;
   db = 0;
```

```
    t = 12; // ... tangle ( degree )
    tStop = 30; // t * tStop = 360 !!!
    pen = 0; // use this for interactivity: pen++
             // if ( pen = tStop )  change the semiaxis !

    Pen( a, 0.0, 0.0 );//first point
    PathX.AddPoint( Pen );
}
```

If two circles glide along two straight lines (Figure 7.5), their centers move on parallel straight lines and therefore satisfy the construction of Figure 7.4 as well. The small wheel of this ellipsograph fixes the pencil and the two bigger wheels, which glide on perpendicular lines. Depending on the positions of the wheels, ellipses, circles, straight lines, or even single points are produced.

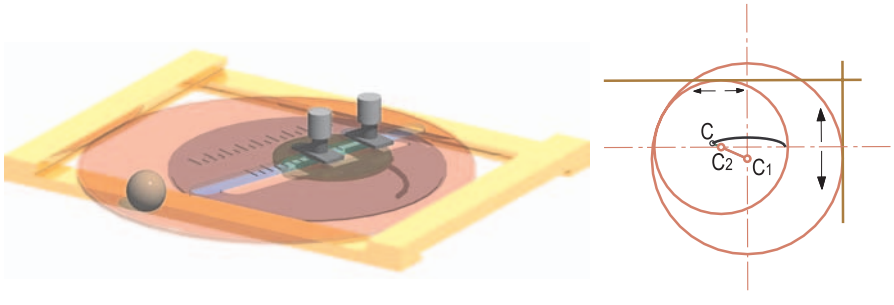

FIGURE 7.5. Wheels gliding along straight lines.

The path curve of the midpoint C_M of the line $\overline{C_1C_2}$ describes a circle with center O and radius $r = \overline{OC_M} = \overline{C_MC_1} = \overline{C_MC_2}$. Due to this property, one slider is replaceable by a crank (Figure 7.6).

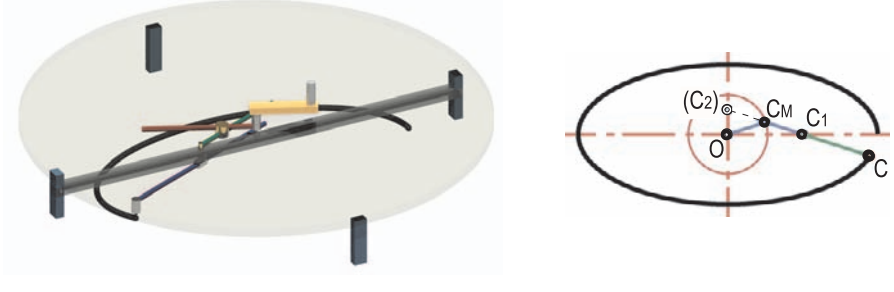

FIGURE 7.6. Generation of an ellipse with a slider crank.

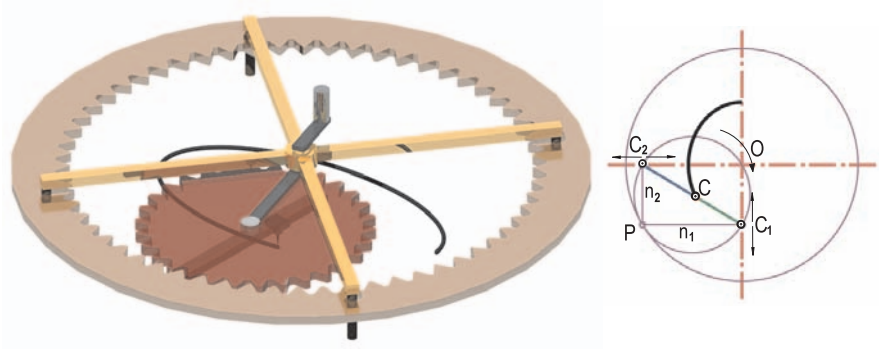

FIGURE 7.7. Inside rolling of two circles.

Planetary motion

The velocity pole P of an arbitrary motion coincides with the normals of the path curves. In our case, we have $P = n_1 \cap n_2$, with n_1 and n_2 being the normals of the path curves of C_1 and C_2. According to $\overline{OP} = \overline{C_1C_2}$ the fixed polode of the motion is a circle with center O and radius $R = \overline{OP}$. The moving polode is a circle as well. Its center is the midpoint C_M of $\overline{C_1C_2}$ and, according to THALES' theorem, its radius r equals $R/2$.

Thus, an ellipse can be generated by rolling a circle inside another circle with radius ratio $R : r = 2 : 1$ (a pair of *Cardan circles*; compare Figure 7.7 and [14], p. 235).

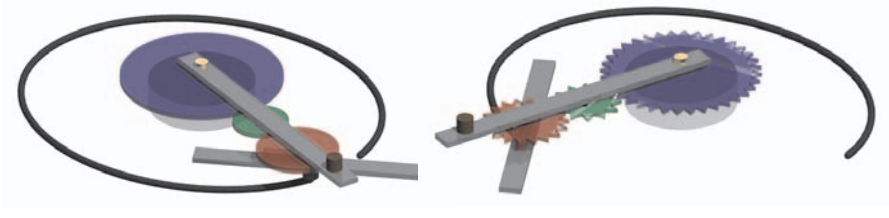

FIGURE 7.8. Planetary motion of gears.

A variation (Figure 7.8) is the *outside rolling* of a circle c_2 of half the radius of the fixed circle c_1. To get the right orientation of revolution, a third circle of arbitrary size is put between them. Each point of the diameter rod of c_2 describes an elliptic path curve.

Inverse elliptic motion

Now let the small Cardan circle be the fixed polode q, while the big circle p rolls outside (Figure 7.9).

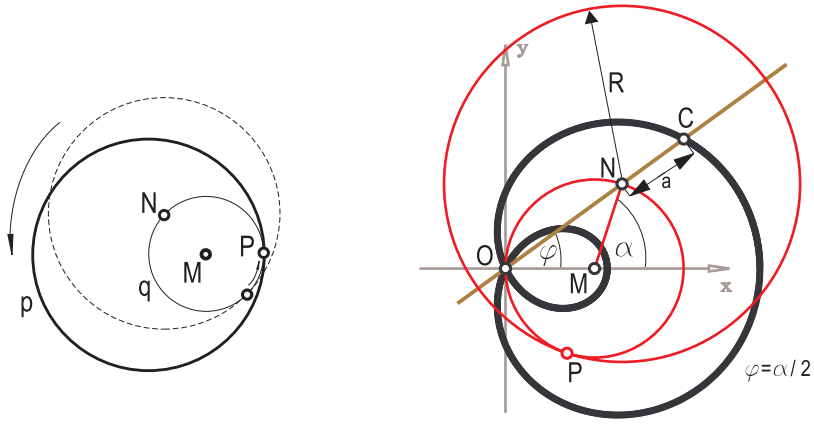

FIGURE 7.9. Inverse elliptic motion (left), limaçon of E. PASCAL (right).

The path curve of points C are limaçons of PASCAL with equation

$$r = a + R\cos\varphi \tag{1}$$

in polar coordinates and

$$\left(x^2 + y^2 - Rx\right)^2 = a^2\left(x^2 + y^2\right)$$

in Cartesian coordinates with $x = r\cos\varphi$, $y = r\sin\varphi$. Here, R is the radius of the moving polode and a the constant distance of point C to the center N of the moving polode. The path is either a curtate ($a < R$), cuspidal ($a = R$), or dimpled ($a > R$) limaçon (Figure 7.9 and Figure 7.10).

Listing from `"X/KH/CONIC_COMPASS/pascal.cpp"`:

```
//Limaçon of Pascal

#include "opengeom.h"

...

void Scene::Animate( )
{
    N.Rotate( M, 1 );
    P.Rotate( M, 1 );
    n.Def( PureRed, N, 2*r );
    nd.Def( N, O );
```

```
    alpha++;
    if ( alpha == 720 )
        alpha = 0;
    phi = alpha / 2;
    phi *= phi * PI / 180;

    //computing a new point of the curve
    C( ( a + 2*r * cos( phi ) ) * cos( phi ),
       ( a + 2*r * cos( phi ) ) * sin( phi ) );

    if ( phi < 360 )
        PathC.AddPoint( C );
}
...
```

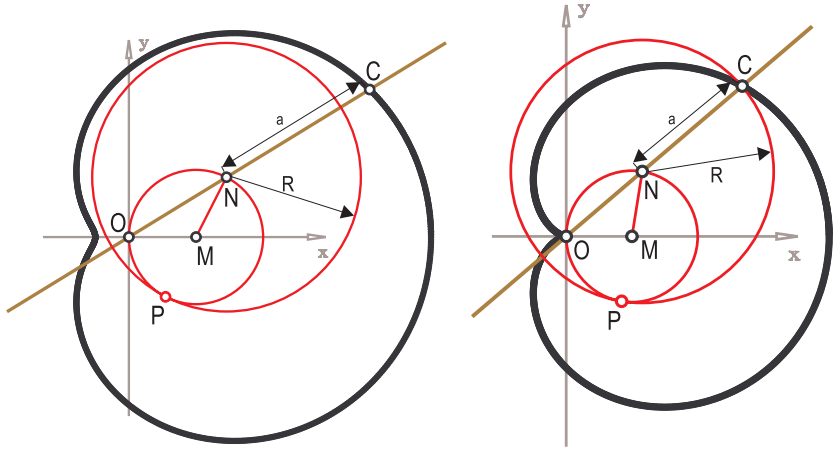

FIGURE 7.10. Dimpled limaçon (left), limaçon with a cusp (cardioid).

Inversor

The mechanism in Figure 7.11 was found by A. Peaucellier (1864) and L. Lipkin (1870). Consisting of four links forming the rhomboid $CA_1C^*A_2$ ($\overline{A_1C} = d_1$) and the links OA_1, OA_2 ($\overline{OA}_1 = \overline{OA}_2 = d_2 \neq d_1$), it rotates about the point O. Consequently the directions A_1A_2 and CC^* will always be perpendicular. Thus, at any position, the points C, C^*, and O lie on a straight line.

Listing from "X/KH/CONIC_COMPASS/InversorPeaucellier.cpp":

```
#include "opengeom.h"

class InvPeau
{
public:
   // Center of inversion; lengths of links
   InvPeau( P2d OIn, Real aIn, Real bIn ); // constructor
   void Init( );
   void ChangeInversor( ); // animation
   void DrawInversor( );
   void SetColors( Color col0In, Color col1In,
                   Color col2In, Color col3In );
private:
   void printHotkeys( );
   P2d O,A,B, C1, C2;
   Real Cx,Cy,c2,r2, a,b, step;
   Circ2d cO,cC, ci;
   Boolean drawPoint;
   PathCurve2d PathC1, PathC2;
   Color col[4];
};

InvPeau ip( Origin, 5, 7 ); // b > a !!!

...

void InvPeau::ChangeInversor( )
{
  int key = TheKeyPressed( );
  switch ( key )
  {
     case 'd': drawPoint = ( drawPoint ) ? false : true;
               break;

     case 'j': if ( ( O.Distance( C1 ) < a + b - 5 * step &&
                      O.Distance( C1 ) > b - a + 2 * step ) ||
                        Cx > A.x )
                                 Cx -= step;
       break; // left

     case 'l': if ( ( O.Distance( C1 ) < a + b - 5 * step &&
         O.Distance( C1 ) > b - a + 2 * step ) ||
                        Cx < A.x )
                                 Cx += step;
       break; // right
```

```
        case 'i': if ( ( O.Distance( C1 ) < a + b − 5 ∗ step &&
                         O.Distance( C1 ) > b − a + 2 ∗ step ) ||
                           Cy < A.y )
                                  Cy += step;
          break; // up

        case 'k': if ( ( O.Distance( C1 ) < a + b − 5 ∗ step &&
                         O.Distance( C1 ) > b − a + 2 ∗ step ) ||
                           Cy > A.y )
                                  Cy −= step;
          break; // down
      }

      cC.Def( Yellow, C1, a );
      cO.SectionWithCircle( cC, A, B );
      r2 = Cx ∗ Cx + Cy ∗ Cy;
      C1( Cx, Cy );
      C2( Cx ∗ c2 / r2, Cy ∗ c2 / r2 );

      if ( drawPoint )
      {
        PathC1.AddPoint( C1 );
        PathC2.AddPoint( C2 );
      }
    }

    ...
```

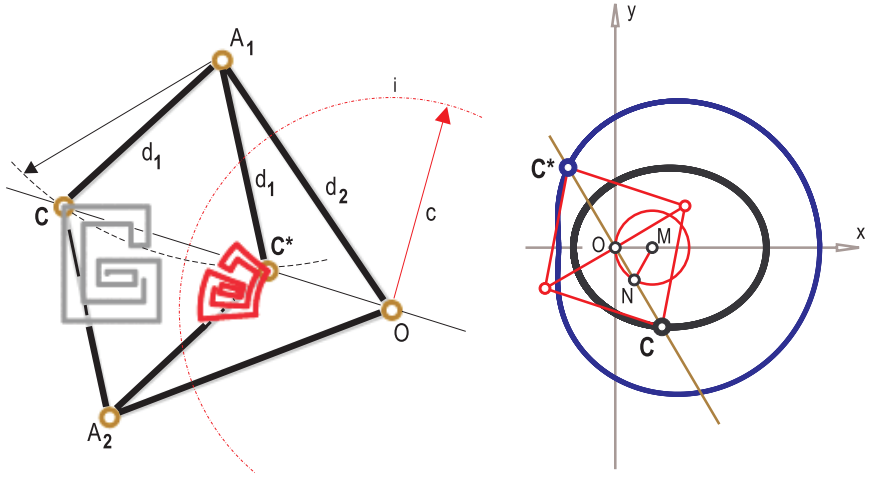

FIGURE 7.11. Inversor of PEAUCELLIER (left), ellipse (right).

The points C and C^* lie on the circle with center A_1 and radius d_1. Thus, we have

$$\overline{OC} \cdot \overline{OC^*} = (d_2 + d_1)(d_2 - d_1) = c^2 \ldots \text{constant},$$

or, in polar coordinates,

$$r^* = \frac{c^2}{r}, \tag{2}$$

with $r = \overline{OC}$ and $r^* = \overline{OC^*}$. This condition is called inversion at a circle i with center O and radius c (Figure 7.11). A mechanism achieving transformation (2) is called an inversor.

Conic compass

The equation

$$r = \frac{p}{1 + e\cos\varphi}$$

describes a conic with O as one focal point and eccentricity e. According to (2),

$$r* = \frac{c^2}{p}(1 + e\cos\varphi)$$

is the equation of its inverse curve. Comparing with (1), we see that this curve is a limaçon of PASCAL.

Thus, combining mechanisms of *inverse elliptic motion* and *inversion* will finally lead to a conic compass. Figure 7.11 and Figure 7.12 show the results of combining a slider crank with PEAUCELLIER's inversor. With $d = \overline{NC^*}$ and $s = \overline{MN}$, the resulting path curve is an ellipse ($d > 2s$), a parabola ($d = 2s$), or a hyperbola ($d < 2s$).

Listing from `"X/KH/H/conic_compass.h"`:

```
...

#ifndef __CONIC_COMPASS__

class ConicCompass
{
  public:
    ConicCompass( P2d inO, Real inS, Real inD, Real inRd, Real inRcs );
    void ChangeCompass( );
    void DrawCompass( );
    void SetColors( Color col0In, Color col1In,
                    Color col2In, Color col3In );
    Real GetCSN( );
```

```
        Real GetMN( );
    private:
        P2d C, M, D, CS, N, A1, A2, S1c, S2c,
            Cfix, CSfix, Nfix, S1fix, S2fix;
        StrL2d nd, ndfix;
        Real phi, alpha, rd, rcs, rC, p,
         eps, stick, step, s, d;
        Circ2d m, cs, cd;
        PathCurve2d PathCS, PathC, PathCSimpossible;
        Boolean collExit;
        Color col[4];
};

...

void ConicCompass::DrawCompass( )
{
    ...

    PathCSimpossible.MarkPoints( col[2], 0.03 );
    PathCS.MarkPoints( col[1], 0.05 );
    PathC.MarkPoints( col[0], 0.07 );

    ...
}

void ConicCompass::ChangeCompass( )
{
    S1c = A1;
    S2c = A2;
    N.Rotate( M, step );
    nd.Def( N, D );
    alpha += step;
    if ( alpha >= 720 ) alpha -= 720;
    phi=alpha / 2;
    CS( D.x + ( d + 2 * s * cos( phi * PI / 180 ) ) * cos( phi * PI / 180 ),
    D.y + ( d + 2 * s * cos( phi * PI / 180 ) ) * sin( phi * PI / 180 ) );

    cs.Def( Yellow, CS, rcs );
    cd.Def( Yellow, D, rd );

    cs.SectionWithCircle( cd, A1, A2 );

    rC = p / ( 1 + eps * cos( phi * PI / 180 ) );
    C( D.x + rC * cos( phi * PI / 180 ), D.y + rC * sin( phi * PI / 180 ) );

    if ( A1 != S1c && phi < 360 )
    {
    PathC.AddPoint( C );
```

```
    PathCS.AddPoint( CS );
    collExit = true;
    }
    else
    {
        if ( collExit )
    {
        Cfix = C;
        CSfix = CS;
        Nfix = N;
        S1fix = A1;
        S2fix = A2;
        ndfix.Def( Nfix, D );
    }

    PathCSimpossible.AddPoint( CS );
    collExit = false;
    }
}

ConicCompass::ConicCompass( P2d inO, Real inS,
        Real inD, Real inRd, Real inRcs )
{
    ...
}

#define __CONIC_COMPASS__
#endif  __CONIC_COMPASS__
```

Listing of "X/KH/CONIC_COMPASS/conics.cpp":

```
#include "opengeom.h"

#include "X/KH/H/conic_compass.h"

P2d C1( -2, 5 ), C2( -7, -5 ), C3( 7, -5 );

ConicCompass e( C1,  // Center
      1.5, // radius s of the circle with center M
           5.3, // distance d from CS to N
         // d < 2s...hyperbola,
         // d = 2s...parabola,
         // d > 2s...ellipse
           3.3, 5.0); // lengths of the links

ConicCompass h( C2, 1.5, 0.7, 3.0, 5.0 ),
```

```
        p( C3, 1.5, 3.0, 3.0, 5.0 );

void Scene::Init( )
{
  e.SetColors( Black, Blue, DarkGray, Red );
  h.SetColors( Black, Magenta, DarkGray, Red );
  p.SetColors( Black, Green, DarkGray, Red );
}

void Scene::Draw( )
{
  e.DrawCompass( );
  PrintString( Black, -8.0, 7.0, "MN =%.2f", e.GetMN( ) );
  PrintString( Black, -8.0, 8.0, "CsN =%.2f", e.GetCSN( ) );
  PrintString( Black, -9.0, 9.0, "ellipse:");

  h.DrawCompass( );
  PrintString( Black, -4.0, -10.0, "MN =%.2f", h.GetMN( ) );
  PrintString( Black, -4.0, -9.0, "CsN =%.2f", h.GetCSN( ) );
  PrintString( Black, -5.0, -8.0, "hyperbola:");

  p.DrawCompass( );
  PrintString( Black, 11.0, -1.0, "MN =%.2f", p.GetMN( ) );
  PrintString( Black, 11.0, 0.0, "CsN =%.2f", p.GetCSN( ) );
  PrintString( Black, 10.0, 1.0, "parabola:");
}

void Scene::Animate( )
{
  e.ChangeCompass( );
  h.ChangeCompass( );
  p.ChangeCompass( );
}

void Scene::CleanUp( )
{
}
```

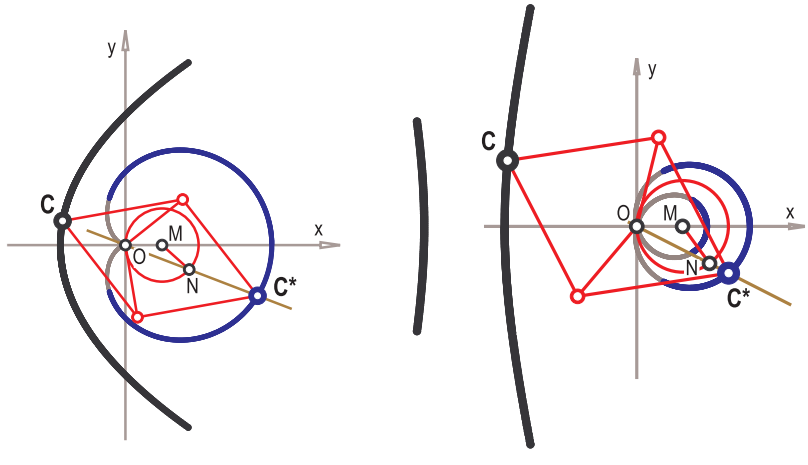

FIGURE 7.12. Mechanism for drawing conic sections (parabola and hyperbola).

This page intentionally left blank

Appendix A

OpenGL Function Reference

The graphics routines of Open Geometry are based on OpenGL. In order to use Open Geometry you need do not need to know anything about OpenGL, but still it may be interesting to get a more profound insight in the background of the Open Geometry class library.

If you want to use Open Geometry for a certain specific task, you might feel the need to adapt some of its drawing and shading routines. In this case you will need to know a few basics about OpenGL. Therefore, we give a very short description of the most important OpenGL functions. Of course, this list is far from complete. If you need more information, please refer to "pure" OpenGL books (e.g., [1], [17], or [36]).

We have sorted the routines according to the following groups:

1. Coordinate Transformations
   glFrustum, glLoadIdentity, glLoadMatrix, glMatrixMode, glMultMatrix, glPopMatrix, glPushMatrix, glRotate, glScale, glTranslate
2. Primitives (Lines, Polygons, Points, ...)
   glBegin, glCullFace, glEdgeFlag, glEnd, glFrontFace, glGetPolygonStripple, glLineStripple, glLineWidth, glPointSize, glPolygonMode, glPolygonStripple, glVertex
3. Color
   glClearColor, glClearIndex, glColor, glColorv, glColorMask, glIndex, glIndexMask, glLogicOp, glShadeModel

4. Lighting
   glColorMaterial, glGetMaterial, glGetLight, glLight, glLightModel, glMaterial, glNormal3

5. Texture Mapping
   glTexCoord, glTexEnv, glTexGen, glTexImage2D, glTexImage1D, glTexParameter, glEnable

6. Raster Graphics
   glCopyPixels, glDrawPixels, glPixelMap, glPixelStore, glPixelTransfer, glPixelZoom, glReadPixels, glRasterPos

A.1 Coordinate Transformations

Whenever you write OpenGL code, it is important to have an understanding of how coordinate transformations in OpenGL work. In particular, you have to think about the order of the transformations if you apply several in a row. Always keep in mind that OpenGL transformations will not change the coordinates of your objects, whereas many Open Geometry-transformations will actually change them.

void *glFrustum*(*GLdouble* left, *GLdouble* right,
GLdouble bottom, *GLdouble* top,
GLdouble near, *GLdouble* far);
◇ Multiplies the current matrix by a matrix
◇ for a perspective-view frustum.

left, right define left and right clipping planes
bottom, top define bottom and top clipping planes
near, far distance to the near and far clipping planes

void *glLoadIdentity*(**void**);
◇ Sets the current matrix to Identity.

void *glLoadMatrix*<d,f>(**const** TYPE* m);
◇ Sets the current matrix to the one specified.
m 4×4 matrix

void *glMatrixMode*(*GLenum* mode);

◇ Specifies the current matrix mode.
mode can be *GL_MODELVIEW*, GL_PROJECION, GL_TEXTURE

void *glMultMatrix*<d,f>(TYPE *m) ;
◇ Multiplies the current matrix by the one specified.
m 4×4 matrix

void *glPopMatrix*(**void**) ;
◇ Pops the current matrix off the matrix stack.

void *glPushMatrix*(**void**) ;
◇ Pushes the current matrix onto the matrix stack.

void *glRotate*<d,f>(TYPE angle, TYPE x, TYPE y, TYPE z) ;
◇ Rotates the current matrix by a rotation matrix.

angle specifies the angle of rotation in degrees
x,y,z vector from the origin that is used as the axis of rotation.

void *glScale*<d,f>(TYPE x, TYPE y, TYPE z) ;
◇ Multiplies the current matrix by a matrix that
◇ scales an object along the x,y,z axes.
x,y,z Scale factors along the x,y, and z axes

void *glTranslate*<d,f>(TYPE x, TYPE y, TYPE z) ;
◇ Multiplies the current matrix by a matrix that
◇ translates an object along the axes.
x,y,z translation vector

A.2 Primitives

OpenGL offers only a few geometrical primitives to be drawn: lines, polygons, and points. The auxiliary libraries provide the programmer with some other "primitives" like spheres and teapots.

void *glBegin*(*GLenum* mode);
◇ Sets the beginning of a group of vertices
◇ that build one or more primitives.

mode	type of primitive operation
GL_POINTS	individual points
GL_LINES	simple lines
GL_LINE_STRIP	series of connected lines
GL_LINE_LOOP	same as above with first and last vertices connected
GL_TRIANGLES	simple triangle
GL_TRIANGLES_FAN	linked fan of triangles
GL_TRIANGLES_STRIP	linked strip of triangles
GL_POLYGON	convex polygon
GL_QUADS	four-sided polygon
GL_QUAD_STRIP	linked strip of four-sided polygons

void *glCullFace*(*GLenum* mode);
mode *GL_FRONT*, *GL_BACK*, *GL_FRONT_AND_BACK*;

void *glEdgeFlag*<v>(*GLboolean*<**const** *GLboolean* *>flag);
◇ Indicates whether a vertex should be considered
◇ as initializing a boundary edge of a polygon.
flag Sets the edge flag to this value.
GL_TRUE (default) or GL_FALSE

void *glEnd*( );
◇ Terminates a group of vertices specified by *glBegin*( ).

void *glFrontFace*(*GLenum* mode);
◇ Defines the front and back sides of a polygon.
mode *GL_CCW* faces with counterclockwise orientation
are considered front-facing. (default)
GL_CW faces with clockwise orientation
are considered front-facing.

void *glGetPolygonStipple*(*GLubyte* *mask);
◇ Returns the stipple pattern of a polygon.

*mask Pointer to the polygon stipple pattern.

void *glLineStipple*(*GLint* factor, *GLushort* pattern);
◇ Specifies the stipple pattern for *GL_LINE...*
◇ primitive operations. First enable strippling
◇ by calling *glEnable*(*GL_LINE_STIPPLE*).

factor The pattern is stretched out by this factor.
pattern Sets the 16-bit long strippling pattern.

void *glLineWidth*(*GLfloat* width);
◇ Sets the current line width.
width width of line in pixels (default is 1.0)

void *glPointSize*(*GLfloat* size);
◇ Sets the current point size.
size size of point (default is 1.0)

void *glPolygonMode*(*GLenum* face, *GLenum* mode);
◇ Sets the drawing mode for the front faces and back faces of a polygon.

face Specifies which faces are affected by
the mode change.
GL_FRONT, *GL_BACK*, *GL_FRONT_AND_BACK*
mode can be *GL_POINT*, *GL_LINE*, *GL_FILL*

void *glPolygonStipple*(**const** *GLubyte** mask)
◇ Specifies the stipple pattern used to fill polygons.
*mask pointer to a 32 * 32-bit array that
contains the stipple pattern.

void glVertex<2,3,4><s,i,f,d><v>(TYPE coords)
◇ Specifies the 3D coordinates of a vertex.
◇ Example: glVertex3f(1.0 , 1.5 , 4.0);

coords can be 1–4 dimensional (x,y,z,w)
w coordinate for scaling purposes

(default is 1.0)

void *glClearColor*(*GLclampf* red, *GLclampf* green,
GLclampf blue, *GLclampf* alpha)
◇ Specifies the current clear color for glClear.

void *glClearIndex*(*GLfloat* index);
◇ Specifies the clearing value for the color index buffer.
index clear color value

void *glColor*<3,4><b,i,s,u,d,f>(TYPE red, TYPE green,
TYPE blue, TYPE alpha);
◇ Sets the current drawing color.

void *glColor*<3,4><b,i,s,u,d,f>v(**const** TYPE *v);
◇ Sets the current drawing color.
v points to an array with four elements

void *glColorMask*(*GLboolean* red, *GLboolean* green,
GLboolean blue, *GLboolean* alpha);
◇ Defines the current mask used to control
◇ writing in RGBA mode.

void *glIndex*<i,s,d,f><v>(TYPE index);
◇ Sets the current drawing index color.
index color index;

void *glIndexMask*(*GLuint* mask);
◇ Sets the mask used to control writing into the color
◇ index buffer by protecting individual bits from being set.
mask bit mask;

void *glLogicOp*(*GLenum* opcode);

◇ Defines the current logical pixel operation for color
◇ index mode.
opcode can be *GL_CLEAR*, *GL_COPY* (**default**), *GL_NOOP*, *GL_SET*, *GL_COPY_INVERTED*, *GL_INVERT*, *GL_AND*, *GL_OR*, *GL_NOR*, *GL_XOR*, *GL_EQUIV*, *GL_AND_INVERTED*, *GL_OR_INVERTED*;

void *glShadeModel*(*GLenum* mode);
◇ Specifies the current shade model
mode *GL_FLAT* or *GL_SMOOTH* (**default**);

A.3 Color

OpenGL allows you to set RGB colors. Furthermore, it enables the programmer to set α-values for transparency.

void *glClearColor*(*GLclampf* red, *GLclampf* green, *GLclampf* blue, *GLclampf* alpha)
◇ Specifies the current clear color for glClear.

void *glClearIndex*(*GLfloat* index);
◇ Specifies the clearing value for the color index buffer.
index clear color value

void *glColor*<3,4><b,i,s,u,d,f>(TYPE red, TYPE green, TYPE blue, TYPE alpha);
◇ Sets the current drawing color.

void *glColor*<3,4><b,i,s,u,d,f>v(**const** TYPE *v);
◇ Sets the current drawing color.
v points to an array with four elements

void *glColorMask*(*GLboolean* red, *GLboolean* green, *GLboolean* blue, *GLboolean* alpha);

◇ Defines the current mask used to control
◇ writing in RGBA mode.

void *glIndex*<i,s,d,f><v>(TYPE index);
◇ Sets the current drawing index color.
index color index

void *glIndexMask*(*GLuint* mask);
◇ Sets the mask used to control writing into the color
◇ index buffer by protecting individual bits from being set.
mask bit mask

void *glLogicOp*(*GLenum* opcode);
◇ Defines the current logical pixel operation for color
◇ index mode.
opcode can be *GL_CLEAR*, *GL_COPY* (**default**), *GL_NOOP*, *GL_SET*, *GL_COPY_INVERTED*, *GL_INVERT*, *GL_AND*, *GL_OR*, *GL_NOR*, *GL_XOR*, *GL_EQUIV*, *GL_AND_INVERTED*, *GL_OR_INVERTED*;

void *glShadeModel*(*GLenum* mode);
◇ Specifies the current shade model
mode *GL_FLAT* or *GL_SMOOTH* (**default**);

A.4 Lighting

When it comes to lighting, OpenGL offers a palette of functions that allow you to use a number of light sources and to provide the objects with physical properties like shininess, etc. Remember that Open Geometry does not use those functions by default, and that it is up to the programmer to introduce the corresponding code.

void *glColorMaterial*(*GLenum* face, *GLenum* mode);
◇ Allows material colors to track the color set by glColor.

face — *GL_FRONT*, *GL_BACK* or *GL_FRONT_AND_BACK*
mode — *GL_EMISSION*, *GL_AMBIENT*, *GL_DIFFUSE*, *GL_SPECULAR* or *GL_AMBIENT_AND_DIFFUSE*

void *glGetMaterial*<i,f>v(*GLenum* face, *GLenum* pname, TYPE param)
◇ Returns the current material property settings.

face — *GL_FRONT*, *GL_BACK* or *GL_FRONT_AND_BACK*

pname:	param:
GL_EMISSION	RGBA values
GL_AMBIENT	RGBA values
GL_DIFFUSE	RGBA values
GL_SPECULAR	RGBA values
GL_SHININESS	specular exponent
GL_COLOR_INDEXES	3 values (ambient, diffuse, specular components)

void *glGetLight*<i,f>v(*GLenum* light, *GLenum* pname, TYPE *params);
◇ Returns information about the current light source settings.

light — light-source *GL_LIGHT0* to *GL_LIGHT7*
pname — specifies which information about the light source is being queried
params — array of integer or floating-point where the return values are stored

pname:	params:
GL_AMBIENT	RGBA values
GL_DIFFUSE	RGBA values
GL_SPECULAR	RGBA values
GL_POSITION	x,y,z coordinates of light source plus one element (usually 1.0)
GL_SPOT_DIRECTION	direction vector for spot light
GL_SPOT_EXPONENT	spot exponent
GL_SPOT_CUTOFF	cutoff angle of the spot source

GL_SPOT_CONSTANT_ATTENUATION	constant attenuation
GL_SPOT_LINEAR_ATTENUATION	linear attenuation
GL_SPOT_QUADRATIC_ATTENUATION	quadratic attenuation

void *glLight*<i,f><v>(*GLenum* light, *GLenum* pname, TYPE param);

◇ Sets the parameters of a light source.

light	light source *GL_LIGHT0* to *GL_LIGHT7*
pname	defines which lighting parameter is to be set (see glGetlight-pname)
param	one or more values that are required to set the lighting parameter

void *glLightModel*<i,f><v>(*GLenum* pname, TYPE param);

◇ Sets the lighting model parameters.

pname: model parameter	params: value(s) for model parameter
GL_LIGHT_MODEL_AMBIENT	array that points to four RGBA components
GL_LIGHT_MODEL_LOCAL_VIEWER	
GL_LIGHT_MODEL_TWO_SIDED	0.0 indicates that only fronts of polygons are included in illumination calculations

void *glMaterial*<i,f><v>(*GLenum* face, *GLenum* pname, TYPE param);

◇ Sets the material parameters.

face: *GL_FRONT*, *GL_BACK* or *GL_FRONT_AND_BACK*

pname:	param:
GL_EMISSION	RGBA values
GL_AMBIENT	RGBA values
GL_DIFFUSE	RGBA values
GL_SPECULAR	RGBA values
GL_AMBIENT_AND_DIFFUSE	RGBA values
GL_SHININESS	specular exponent (1 value)
GL_COLOR_INDEXES	3 values (ambient, diffuse, specular components)

void *glNormal3*<b,i,s,d,f>(TYPE nx, TYPE ny, TYPE nz);
◇ Defines a normal vector whose direction is up and
◇ perpendicular to the surface of the polygon.
nx, ny, nz x, y, z magnitudes of the normal vector

void *glNormal3*<b,i,s,d,f>v(**const** TYPE *v);
v array of three elements where
normal vector is stored

A.5 Texture Mapping

Texture mapping is one of the best features that OpenGL offers. If the mapping is done by hardware, one can produce realistic images in a very short time.

void *glTexCoord*<1,2,3,4><dfis>(TYPE s , TYPE t ,
TYPE r , TYPE q);
◇ Specifies the current texture image coordinates (1D–4D)
◇ for the following vertex of a textured polygon.

s,t x,y coordinates on texture image (0.0 – 1.0)
r texture image depth coordinate
q texture image "time" coordinate

void *glTexEnv*<f,i,fv,iv> (*GLenum* target ,
GLenum pname , TYPE);
◇ Sets parameters for texture-mapping environment.

target must be *GL_TEXTURE_ENV*

pname:	param:
GL_TEXTURE_ENV_COLOR	param is a pointer to an RGBA color value.
GL_TEXTURE_ENV_MODE	to define type of texture mapping

– *GL_DECAL* texture is directly mapped
– *GL_BLEND* texture is blended by constant color
– *GL_MODULATE* texture is shaded

void *glTexGen*<d,f,i,dv,fv,iv> (*GLenum* coord, *GLenum* pname,
TYPE param) ;
◇ Defines parameters for automatic
◇ texture coordinate generation.

coord must be *GL_S*, *GL_T*, *GL_R* or *GL_Q*

pname:	param:
GL_TEXTURE_GEN_MODE	– *GL_OBJECT_LINEAR* – *GL_EYE_LINEAR* – *GL_SPHERE_MAP*
GL_OBJECT_PLANE	
GL_EYE_PLANE	

void *glTexImage2D*(*GLenum* target, *GLint* level,
GLint components, *GLsizei* width,
GLsizei height, *GLint* border, *GLenum* format,
GLenum type, **const** *GLvoid* *pixels) ;
◇ Describes a two-dimensional texture image

target	must be *GL_TEXTURE_2D*
level	usually 0 (these are multiple texture resolutions (mip mapping))
width	texture image width (must be a power of 2) + 2*border
height	texture image height (must be a power of 2) + 2*border
border	width of border (0,1, or 2)
format	usually *GL_RGB*(A) or *GL_COLOR_INDEX*, *GL_RED*, *GL_GREEN*, *GL_BLUE*, *GL_ALPHA*, *GL_LUMINANCE* (grayscale), *GL_ALPHA_LUMINANCE*
type	data type of pixel value usually *GL_UNSIGNED_BYTE*, *GL_BYTE*, *GL_BITMAP*, *GL_SHORT*, *GL_UNSIGNED_SHORT*, *GL_INT*, *GL_UNSIGNED_INT*, *GL_FLOAT*
pixels	array for pixel data

void *glTexImage1D*(*GLenum* target, *GLint* level, *GLint* components,
GLsizei width, *GLint* border, *GLenum* format,
GLenum type, **const** *GLvoid* *pixels) ;
◇ Describes a one-dimensional texture image

target must be *GL_TEXTURE_1D*

see *glTexImage2D*

void *glTexParameter*<f,fv,i,iv>(*GLenum* target, *GLenum* pname,
TYPE param);
◇ Defines how the texture image is mapped onto the polygon.

target must be *GL_TEXTURE_1D* or *GL_TEXTURE_2D*

pname:	param:	
mapping parameters		
GL_TEXTURE_WRAP_S	*GL_CLAMP*	to clamp texture
GL_TEXTURE_WRAP_T	*GL_REPEAT*	repeat texture until the end of the polygon
GL_TEXTURE_MAG_FILTER	*GL_NEAREST*	no filtering
GL_TEXTURE_MIN_FILTER	*GL_LINEAR*	filtering
GL_TEXTURE_BORDER_COLOR	RGBA value	

for mip mapping use
GL_NEAREST<LINEAR>_MIPMAP_NEAREST_<LINEAR>

void *glEnable*(parameter)
◇ This enables texture mapping.
parameter *GL_TEXTURE_2D* or *GL_TEXTURE_1D*

A.6 Raster Graphics

Raster graphics are a powerful link between pure mathematical calculations and the actual drawing window. Nevertheless, we tried to avoid them whenever possible, namely, because Open Geometry must not rely on the screen resolution: For example, if we produce a print of an image, the resolution of the printing device is usually quite different from of the window size.

void *glCopyPixels*(*GLint* x, *GLint* y, *GLsizei* width,
GLsizei height, Glenum type);
◇ Copies a block of pixels into the frame buffer at
◇ a position defined by glRasterPos.

type	*GL_COLOR* to copy color values
	GL_STENCIL to copy stencil values
	GL_DEPTH to copy depth values

void *glDrawPixels*(*GLsizei* width, *GLsizei* height, *GLenum* format,
GLenum type, **const** *GLvoid* *pixels);
◇ Draws a block of pixels into the frame buffer
◇ at a position defined by glRasterPos.

format	usually *GL_RGBA* or *GL_RGB*, *GL_COLOR_INDEX*, *GL_RED*, *GL_GREEN*, *GL_BLUE*, *GL_ALPHA*, *GL_LUMINANCE* (grayscale) , *GL_ALPHA_LUMINANCE*
type	data type of pixel value *GL_UNSIGNED_BYTE*, *GL_BYTE*, *GL_BITMAP*, *GL_SHORT*, *GL_UNSIGNED_SHORT*, *GL_INT*, *GL_UNSIGNED_INT*, *GL_FLOAT*

void *glPixelMap*<fv,uiv,usv>(*GLenum* map, *GLint* mapsize, TYPE values);
◇ Sets a lookup table for *glCopyPixels*, *glDrawPixels*, *glReadPixels*,
◇ *glTexImage2D*, *glTexImage1D*. *GL_MAP_COLOR* or *GL_MAP_STENCIL*
◇ have to be enabled with *glPixelTransfer*.

map:	Define a lookup table for
GL_PIXEL_MAP_I_TO_I	color indices
GL_PIXEL_MAP_S_TO_S	stencil values
GL_PIXEL_MAP_I_TO_R	color indices to red values
GL_PIXEL_MAP_I_TO_G	color indices to green values
GL_PIXEL_MAP_I_TO_B	color indices to blue values
GL_PIXEL_MAP_I_TO_A	color indices to alpha values
GL_PIXEL_MAP_R_TO_R	red values
GL_PIXEL_MAP_G_TO_G	green values
GL_PIXEL_MAP_B_TO_B	blue values
GL_PIXEL_MAP_A_TO_A	alpha values

mapsize	size of lookup table (must be power of 2)
values	pointer to lookup table

void *glPixelStore*<i,f>(*GLenum* pname, TYPE param);
◇ Defines how glDrawPixels, glTexImage1D, glTexImage2D,
◇ glReadPixels read and store pixel data.

pname:	param:	result:
GL_PACK_SWAP_BYTES	*GL_TRUE*	bytes are swapped when stored in memory
GL_UNPACK_SWAP_BYTES	*GL_TRUE*	read from memory
GL_PACK_LSB_FIRST	GL_FALSE	leftmost pixel of bitmap is stored/read in bit 0
GL_UNPACK_LSB_FIRST	GL_FALSE	instead of bit 7
GL_PACK_ROW_LENGTH	integer	
GL_UNPACK_ROW_LENGTH	integer	
GL_PACK_SKIP_PIXELS	integer	
GL_UNPACK_SKIP_PIXELS	integer	
GL_PACK_SKIP_ROWS	integer	
GL_UNPACK_SKIP_ROWS	integer	
GL_PACK_ALIGNMENT	integer	
GL_UNPACK_ALIGNMENT	integer	

void *glPixelTransfer*<i,f>(*GLenum* pname, TYPE param)
◇ Sets pixel-transfer modes that affect the operations
◇ of *glDrawPixels*( ), *glReadPixels*( ), *glCopyPixels*( ),
◇ *glTexImage1D*( ), *glTexImage2D*( ) and glGetTexImage( ).

pname	*GL_MAP_COLOR*, *GL_MAP_STENCIL*, *GL_INDEX_SHIFT*, *GL_INDEX_OFFSET*, *GL_RED_SCALE*, *GL_GREEN_SCALE*, *GL_BLUE_SCALE*, *GL_ALPHA_SCALE*, *GL_DEPTH_SCALE*, *GL_RED_BIAS*, *GL_GREEN_BIAS*, *GL_BLUE_BIAS*, *GL_ALPHA_BIAS*, *GL_DEPTH_BIAS*;
param	*GLint*, *GLfloat* parameter value

void *glPixelZoom*(*GLfloat* xfactor, *GLfloat* yfactor)
◇ Sets pixel scaling factors for pixel-transfer operations.

xfactor, yfactor	pixel scaling factors along the x, y axes

void *glReadPixels*(*GLint* x, *GLint* y, *GLsizei* width,
GLsizei height, *GLenum* format,
GLenum type, *GLvoid* *pixels)
◇ Reads a block of pixel data from the frame buffer
◇ and stores it in the array pointed to by pixels.

format	indicates the kind of pixel data elements that are read *GL_RGBA*, *GL_RGB*, *GL_COLOR_INDEX*, *GL_RED*, *GL_GREEN*, *GL_BLUE*, *GL_ALPHA*, *GL_LUMINANCE* (grayscale) or *GL_ALPHA_LUMINANCE*;
type	indicates the data type of each element *GL_UNSIGNED_BYTE*, *GL_BYTE*, *GL_BITMAP*, *GL_SHORT*, *GL_UNSIGNED_SHORT*, *GL_INT*, *GL_UNSIGNED_INT*, *GL_FLOAT*
pixels	pointer to array

void *glRasterPos*<234><sifd><v>(TYPE x, TYPE y, TYPE z, TYPE w);

◇ Sets the raster position at the specified coordinates.

Appendix B

List of Examples

This page intentionally left blank

Appendix C

Open Geometry Class List

Open Geometry is an *object oriented* geometric library. There exist relatively few base classes that all others are built on. The most important base classes are certainly *O2d* and *O3d*. Roughly speaking, they are used to describe conglomerates of points in two or three dimensions (classes *P2d* and *P3d*). Both, *O2d* and *O3d*, are derived from the class *O23d* that contains of a number of objects of type *P23d*, the common ancestor of *P2d* and *P3d*.[1]

As a result of this approach, many Open Geometry classes are automatically equipped with useful methods concerning object size and color or application of geometric transformations. Compare the corresponding section on page 579 for more information.

The knowledge of the hierarchic structure of Open Geometry is of relevance to the user. It helps finding the appropriate method for a certain programming task. In addition to the detailed descriptions in Chapter 6 we decided to provide an online-help in HTML format.

Just open `"oghelp.html"` with your favorite web browser. It will display three frames that help you finding Open Geometry's important classes and methods. In the leftmost frame, you can see an alphabetic list. Clicking on the desired class or methods scrolls the big main frame to the corresponding declaration[2].

[1] In fact, *O23d* contains much more than just a pointer to a point array. It is already equipped with everything that is necessary for later drawing and shading. In general, however, not all member variables are really initialized.

[2] In general, this is a copy of the original declaration in the header-file of the class.

Note that the main frame contains links to the direct parent class. Scroll, e.g., to the declaration of *Circ3d*. You will see that it is derived from *RegPoly3d*. Click on the link and you will see the declaration of *RegPoly3d*.

Finally, the alphabet in the top frame helps you navigating through the class list in the left frame. The "hierarchy"-link will open an hierarchic class list. The child classes of a certain level are intended by a constant amount and, again, a simple mouse click takes you to the declaration. If you want to switch back to an alphabetic view, just click on an an arbitrary letter in the top frame.

Occasionally, we erased one or the other method that is not intended for public use.

Appendix D

Installation of Open Geometry

Open Geometry does not come as an executable. Except for some demo files, you cannot start it directly from a command prompt or by double clicking on some icon. It is a programming package, and in order to use it, you need a C++ compiler.

The accompanying CD-ROM contains two directories `"WINDOWS/"` and `"LINUX/"`. There you find all necessary components for a Windows environment and a Linux (or Unix) environment:

Windows

Installing Open Geometry in a Windows environment is easy: Just start `"setup.exe"` in the `"WINDOWS/"` directory on the CD-ROM. This will create a local Open Geometry folder (by default `"OPENGEOM/"`) on your computer. From this point on, you can follow the steps described in Section 1.4.

Linux

If you are working in a Linux or Unix environment, proceed as follows:

1. Copy the file into your home directory. You need not create an Open Geometry directory!
2. In order to extract the file `"OpenGeometry2.0.tar.gz"`, please write, e.g.,

```
gzip -d *.tar.gz
tar -cf *.tar
```

This will create a directory "OpenGeometry2.0"

3. Switch to the directory "OpenGeometry2.0"
4. Now you can compile with

   ```
   make x
   ```

   It should be easy to compile the source code. When it comes to the linking process, you might have to install some mesa libraries.

If you run into trouble, please read the file "read.me" on the CD-ROM. If this does not help, you can contact us via email (open.geometry@uni-ak.ac.at).

Updating Open Geometry 1.0

If you want to update a previous version of Open Geometry, you can simply overwrite the existing files. If you have changed one of the files in the "C/" directory, however, we recommend the following procedure:

1. Rename the existing Open Geometry folder (e.g. "OG_10/").
2. Install the new version.
3. Copy your own demo programs into the new "USER/" directory and include them, as usual, via "try.cpp".
4. In order to integrate your own code in the new version, please, refer to page 24 and read the instructions for user contributions.

Okay. Let's assume that you have installed Open Geometry correctly on your system. Then you should try out a demo-executable from the "DEMOS/" directory, e.g., "demo1.exe". There, you can try different buttons and menu items in order to get a feeling for a typical Open Geometry application.

If everything works correctly, the first thing you will see is a welcome message informing you that a default file "og.ini" will be created. There you can set some of Open Geometry's default parameters (compare Section 7.3). However, for the time being you need not care about it. Just learn how to write an Open Geometry application of your own.

References

[1] E. Angle: Interactive Computer Graphics: A Top-Down Approach with OpenGL. Addison-Wesley, 1999.

[2] I. Artobolevskii: Mechanisms of the Generation of Plane Curves. Pergamon Press, Oxford–London, 1964.

[3] E. Blutel: Recherches sur les surfaces qui sont en même temps lieux de coniques et enveloppes de cônes du second degré. Ann. sci. école norm. super., 7(3), 1890, 155–216.

[4] J.W. Bruce, P.J. Giblin, C.G. Gibson: On Caustics of Plane Curves. Am. Math. Month. **88**, 1981, 651–657.

[5] Z. Čerin: Regular Hexagons Associated to Centroid Sharing Triangles. Beiträge zu Algebra und Geometrie (Contributions to Algebra and Geometry) **39** 1998 263-267.

[6] B. Ernst: Magic Mirror of M. C. Escher. TASCHEN America LIc, 1995.

[7] K.J. Falconer: Fractal Geometry. Mathematical Foundations and Applications. John Wiley & Sons Ltd., Chichester 1990.

[8] K. Fladt, A. Baur: Analytische Geometrie spezieller Flächen und Raumkurven. Braunschweig 1975.

[9] F. Granero Rodríguez, F. Jiménez Hernandez, J.J. Doria Iriarte: Constructing a family of conics by curvature-depending offsetting from a given conic. Computer Aided Geometric Design **16**, 1999, 793–815.

[10] A. Gehrer, H. Passrucker, H. Jericha, J. Lang: Blade Design and Grid Generation for Computational Fluid Dynamics with Bézier Curves and Bézier Surfaces. Proceedings of the 2nd European Conference on Turbomachinery — Fluid Dynamics and Thermodynamics, Antwerpen, Belgium, March 5–7, 1997, 273–280.

[11] G. Glaeser: Objektorientiertes Graphik-Programmieren mit der Pascal-Unit Supergraph. B. G. Teubner, Stuttgart, 1992.

[12] G. Glaeser: Von Pascal zu C/C++. Markt&Technik, München, 1993.

[13] G. Glaeser: Fast Algorithms for 3D-Graphics. Springer-Verlag, New York, 1994.

[14] G. Glaeser, H. Stachel: Open Geometry: OpenGL and Advanced Geometry. Springer-Verlag, New York 1999.

[15] G. Glaeser: Reflections on Spheres and Cylinders of Revolution. Journal for Geometry and Graphics (JGG) **2** (2), Heldermann Verlag Berlin, 1999, 1–19.

[16] G. Glaeser, H.P. Schröcker: Reflections on Refractions. J. Geometry Graphics (JGG) **4** (1), Heldermann Verlag Berlin, 2000.

[17] F.J. Hill, F.S. Hill: Computer Graphics Using OpenGL. Prentice Hall, 2000.

[18] J. Hoschek, D. Lasser: Grundlagen der Geometrischen Datenverarbeitung. Stuttgart 1992.

[19] W. Jank: Flächen mit kongruenten Fallparabeln. Sb. sterr. Akad. Wiss. **181** 1973, 49–70.

[20] E. Kruppa: Natürliche Geometrie der Mindingschen Verbiegungen der Strahlflächen. Mh. Math. **55**, 1951.

[21] E.E. Kummer: Gipsmodell der Steinerschen Fläche. Monatsber. Akad. Wiss. Berlin **1863**, 539.

[22] J. Lang, H.P. Schröcker: Edge-Orthogonal Patches through a Given Rational Bezier Curve. Journal for Geometry and Graphics (JGG) **2** (2), Heidemann Verlag Berlin, 1998, 109–121.

[23] B. Mandelbrot: Fractal Geometry of Nature. W.H. Freeman & Company New York, 1977.

[24] F.A. Möbius: Über die Bestimmung des Inhaltes eines Polyeders. Ber. Verh. Sächs. Ges. Wiss. **17**, 1865, 31–68.

[25] E. Müller, J.L. Krames: Vorlesungen über Darstellende Geometrie, Bd. II: Die Zyklographie. Leipzig, Wien: Franz Deuticke, 1929.

[26] E. Müller, J.L. Krames: Vorlesungen über Darstellende Geometrie, Bd. III: Konstruktive Behandlung der Regelflächen. Leipzig, Wien: Franz Deuticke, 1931. S

[27] H. Pottmann, J. Wallner: Computational Line Geometry. Springer-Verlag, Heidelberg 2001.

[28] M.J. Pratt: Dupin cyclides and supercyclides. In: The mathematics of surfaces VI. Glen Mullineux, editor: The Institute of Mathematics and its Applications, Oxford University Press, 1996, 43–66.

[29] M.J. Pratt: Quartic supercyclides I: Basic theory. Computer Aides Geometric Design 14(7), 1997, 671–692.

[30] H. Sachs, G. Karáné: Schall- und Brechungsfronten an ebenen Kurven. Publ. Math. Debrecen **54**, 1999, 189–205.

[31] M. Sadowsky: Theorie der elastisch biegsamen undehnbaren Bänder mit Anwendungen auf das Möbiussche Band. Verh. 3. Intern. Kongr. Techn. Mechanik, Stockholm, 1930.

[32] M. Sadowsky: Ein elementarer Beweis für die Existenz eines abwickelbaren Möbiusschen Bandes und Zurückführung des geometrischen Problems auf ein Variationsproblem. Sitzgsber. Preuss. Akad. Wiss. **22**, 1930, 412–415.

[33] D. Schattschneider: M. C. Escher Kaleidocycles. Pomegranate Artbooks Inc, 1987.

[34] H.P. Schröcker: Die von drei projektiv gekoppelten Kegelschnitten erzeugte Ebenenmenge. Dissertation an der Technischen Universität Graz, 2000.

[35] J. Schwarze: Cubic and Quartic Roots. In Graphics Gems (A. Glassner, ed.), Academic Press, 1990, 404–407.

[36] D. Shreiner: OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2. Addison-Wesley, 1999.

[37] G. Weiss: Räumliche Deutung von Ellipsenkonstruktionen. IBDG 2/1998.

[38] W. Wunderlich: Über ein abwickelbares Möbiusband. Monatshefte für Mathematik **66**, 1962, 276–289.

[39] W. Wunderlich: Über zwei durch Zylinderrollung erzeugbare Modelle der Steiner'schen Römerfläche. Archiv der Mathematik, Vol. XVIII, 1967, 325–336.

[40] W. Wunderlich: Darstellende Geometrie II. Mannheim: Bibliographisches Institut, 1967.

[41] W. Wunderlich: Ebene Kinematik. Mannheim: Bibliographisches Institut, 1970.

Index of Files

Index