Theory of Three-Dimensional Computer Graphics
By L. Szirmay-Kalos

Contents:

Chapter 1

INTRODUCTION

1.1   Synthetic camera model

Suppose that a man is sitting in front of a computer calculating a function over its domain. In its simplest realization the program keeps printing out the samples of the domain with their respective function value in alphanumeric form. The user of the program who is mainly interested in the shape of the function has a very hard time reading all the data before they are scrolled off the screen, interpreting numbers like $1.2345e12$, and constructing the shape in his mind. He would prefer the computer to create the drawing of the function itself, and not to bother him with a list of mind-boggling floating point numbers. Assume that his dream comes true immediately, and the computer draws horizontal rectangles proportional to the function value, instead of printing them out in numeric form, making a histogram-like picture moving up as new values are generated. The user can now see the shape of a portion of the function. But is he satisfied? No. He also wants to have a look at the shape of larger parts of the function; he is not happy with the reduced accuracy caused by the limited resolution of the computer screen, meaning for example that two values that are very close to each other would share the same rectangle, and very large values generate rectangles that run off the screen. It irritates him that if he turns his head for just a second, he loses a great portion of the function, because it has already been scrolled off. The application of graphics instead of a numeric display has not solved many problems.

In order to satisfy our imaginary user, a different approach must be chosen; something more than the simple replacement of output commands by drawing primitives. The complete data should be generated and stored before being reviewed, thus making it possible to scale the rectangles adaptively in such a way that they would not run off the screen, and allowing for response to user control. Should the user desire to examine a very small change in the function for example, he should be able to zoom in on that region, and to move back and forth in the function reviewing that part as he wishes, etc.

This approach makes a clear distinction between the three main stages of the generation of the result, or image. These three stages can be identified as:

- Generation of the data

- Storage of the data

- Display of the data

The components of "data generation" and "data display" are not in any hierarchical relationship; "data display" routines are not called from the "data generation" module, but rather they respond to user commands and read out "data storage" actively if the output of the "data generation" is needed.

The concept which implements the above ideas is called the **synthetic camera model**, and is fundamental in most graphics systems today, especially in the case of three-dimensional graphics. The main components of the synthetic camera model are generalizations of the components in the previous example:

- **Modeling:**

  Modeling refers to a process whereby an internal representation of an imaginary world is built up in the memory of the computer. The modeler can either be an application program, or a user who develops the model by communicating with an appropriate software package. In both cases the model is defined by a finite number of applications of primitives selected from finite sets.

- **Virtual world representation:**

  This describes what the user intended to develop during the modeling phase. It can be modified by him, or can be analyzed by other programs capable of reading and interpreting it. In order to allow for easy modification and analysis by different methods, the model has to represent all relevant data stored in their natural dimensions. For example, in an architectural program, the height of a house has to be represented in meters, and not by the number of pixels, which would be the length of the image of the house on the screen. This use of natural metrics to represent data is known as the application of a **world coordinate system**. It does not necessarily mean that all objects are defined in the very same coordinate system. Sometimes it is more convenient to define an object in a separate, so-called **local coordinate system**, appropriate to its geometry. A transformation, associated with each object defining its relative position and orientation, is then used to arrange the objects in a common global **world coordinate system**.

- **Image synthesis:**

  Image synthesis is a special type of analysis of the internal model, when a photo is taken of the model by a "software camera". The position and direction of the camera are determined by the user, and the image thus generated is displayed on the computer screen. The user is in control of the camera parameters, lightsources and other studio objects. The ultimate objective of image synthesis is to provide the illusion of watching the real objects for the user of the computer system. Thus, the color sensation of an observer watching the artificial image generated by the graphics system about the internal model of a virtual world must be approximately equivalent to the color perception which would be obtained in the real world. The color perception of humans depends on the shape and the optical properties of the objects, on the illumination and on the properties and operation of the eye itself. In order to model this complex phenomenon both the physical-mathematical structure of the light-object interaction and the operation of the eye must be understood. Computer screens can produce controllable electromagnetic waves, or colored light for their

observers. The calculation and control of this light distribution are the basic tasks of image synthesis which uses an internal model of the objects with their optical properties, and implements the laws of physics and mathematics to simulate real world optical phenomena to a given accuracy. The exact simulation of the light perceived by the eye is impossible, since it would require endless computational process on the one hand, and the possible distributions which can be produced by computer screens are limited in contrast to the infinite variety of real world light distributions on the other hand. However, color perception can be approximated instead of having a completely accurate simulation. The accuracy of this approximation is determined by the ability of the eye to make the distinction between two light distributions. There are optical phenomena to which the eye is extremely sensitive, while others are poorly measured by it. (In fact, the structure of the human eye is a result of a long evolutionary process which aimed to increase the chance of survival of our ancestors in the harsh environment of the pre-historic times. Thus the eye has become sensitive to those phenomena which were essential from that point of view. Computer monitors have had no significant effect on this process yet.) Thus image synthesis must model accurately those phenomena which are relevant but it can make significant simplifications in simulating those features for which the eye is not really sensitive.

This book discusses only the image synthesis step. However, the other two components are reviewed briefly, not only for the reader's general information, but also that model dependent aspects of image generation may be understood.

1.2 Signal processing approach to graphics

From the information or signal processing point of view, the modeling and image synthesis steps of the synthetic camera model can be regarded as transformations (figure 1.1). Modeling maps the continuous world which the user intends to represent onto the discrete internal model. This is definitely an analog-digital conversion. The objective of image synthesis is the generation of the data analogous to a photo of the model. This data
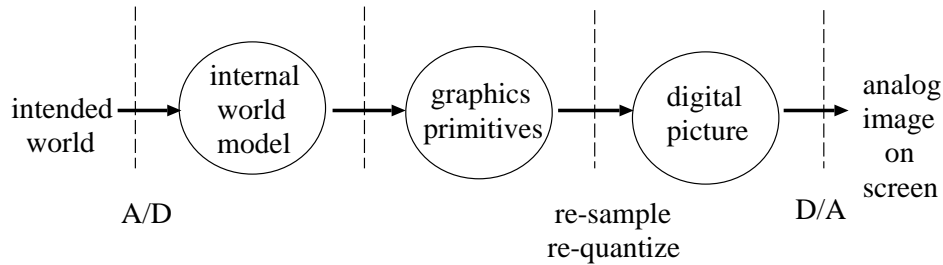
Figure 1.1: Dataflow model of computer graphics

is stored in the computer and is known as digital picture, which in turn is converted to analog signals and sent to the display screen by the computer's built in graphics hardware.

The digital picture represents the continuous two-dimensional image by finite, digital data; that is, it builds up the image from a finite number of building blocks. These building blocks can be either one-dimensional, such as line segments (called **vectors**), or two-dimensional, such as small rectangles of uniform color (called **pixels**). The word *"pixel"* is a composition of the words *"picture"* and *"element"*.

The digital picture represented by either the set of line segments or the pixels must determine the image seen on the display screen. In **cathode ray tube (CRT)** display technology the color and the intensity of a display point are controlled by three electron beams (exciting red, green and blue phosphors respectively) scanning the surface of the display. Thus, the final stage of graphics systems must convert the digital image stored either in the form of vectors or pixels into analog voltage values used to control the electron beams of the display. This requires a digital-analog conversion.

1.3   Classification of graphics systems

The technique of implementing vectors as image building blocks is called **vector graphics**. By the application of a finite number of one-dimensional primitives only curves can be generated. Filled regions can only be approximated, and thus vector graphics is not suitable for realistic display of solid objects. One-dimensional objects, such as lines and characters defined as

a list of line segments and jumps between these segments, are represented by relative coordinates and stored in a so-called **display list** in a vector graphics system. The end coordinates of the line segments are interpreted as voltage values by a **vector generator** hardware which integrates these voltages for a given amount of time and controls the electron **beam** of the cathode ray tube by these integrated voltage values. The beam will draw the sequence of line segments in this way similarly to electronic oscilloscopes. Since if the surface of the display is excited by electrons then it can emit light only for a short amount of time, the electron beam must draw the image defined by the display list periodically about 30 times per second at least to produce **flicker-free** images.

Raster graphics, on the other hand, implements pixels, that is two-dimensional objects, as building blocks. The image of a raster display is formed by a **raster mesh** or **frame** which is composed of horizontal *raster* or **scan-lines** which in turn consist of rectangular pixels. The matrix of pixel data representing the entire screen area is stored in a memory called the **frame buffer**. These pixel values are used to modulate the intensities of the three electron beams which scan the display from left to right then from top to bottom. In contrast to vector systems where the display list controls the direction of the electron beams, in raster graphics systems the direction of the movement of the beams is fixed, the pixel data are responsible only for the modulation of their intensity. Since pixels cover a finite 2D area of the display, filled regions and surfaces pose no problem to raster based systems. The number of pixels is constant, thus the cycle time needed to avoid flickering does not depend on the complexity of the image unlike vector systems. Considering these advantages the superiority of raster systems is nowadays generally recognized, and it is these systems only that we shall be considering in this book.

When comparing vector and raster graphics systems, we have to mention two important disadvantages of raster systems. Raster graphics systems store the image in the form of a pixel array, thus normal image elements, such as polygons, lines, 3D surfaces, characters etc., must be transformed to this pixel form. This step is generally called the **scan conversion**, and it can easily be the bottleneck in high performance graphics systems. In addition to this, due to the limitations of the resolution and storage capability of the graphics hardware, the digital model has to be drastically re-sampled and re-quantized during image generation. Since the real or

intended world is continuous and has infinite bandwidth, the Shannon–Nyquist criterion of correct digital sampling cannot be guaranteed, causing artificial effects in the picture, which is called **aliasing**.

Note that scan-conversion of raster graphics systems transforms the geometric information represented by the display list to pixels that are stored in the frame buffer. Thus, in contrast to vector graphics, the display list is not needed for the periodic screen refresh.

1.4 Basic architecture of raster graphics systems

A simple raster graphics system architecture is shown in figure 1.2. The **display processor** unit is responsible for interfacing the frame buffer memory with the general part of the computer and taking and executing the drawing commands. In personal computers the functions of this display processor are realized by software components implemented in the form of a graphics library which calculates the pixel colors for higher level primitives. The programs of this graphics library are executed by the main CPU of the computer which accesses the frame buffer as a part of its operational memory.
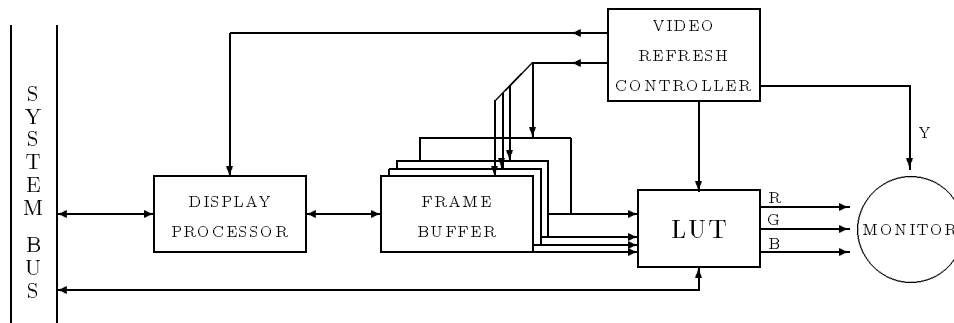

Figure 1.2: Raster system architecture with display processor

In advanced systems, however, a special purpose CPU is allocated to deal with pixel colors and to interface the frame buffer with the central CPU. This architecture increases the general performance because it relieves the central CPU of executing time consuming scan-conversion tasks on the one hand, and makes it possible to optimize this display processor for the graphics tasks on the other hand. The central CPU and the display processor communicate using drawing commands referring to higher level graphics primitives. The level of these primitives and the coordinate system where their geometry is defined is a design decision. These primitives are then transformed into pixel colors by the display processor having executed the image synthesis tasks including transformations, clipping, scan-conversion etc., and finally the generated pixel colors are written into the frame buffer memory. Display processors optimized for these graphics operations are called **graphics (co)processors**. Many current graphics processor chips combine the functions of the display processor with some of the functions of the video refresh controller, as for example the TMS 34010/20 [Tex88] from Texas Instruments and the HD63484 [Hit84] from Hitachi, allowing for compact graphics architectures. Other chips, such as i860 [Int89] from Intel, do not provide hardware support for screen refresh and timing, thus they must be supplemented by external refresh logic.

The **frame buffer memory** is a high-capacity, specially organized memory to store the digital image represented by the pixel matrix. For each elemental rectangle of the screen — that is for each pixel — a memory word is assigned in the frame buffer defining the color. Let the number of bits in this word be n. The value of n is 1 for **bi-level** or **black-and-white devices**, 4 for cheaper color and gray-shade systems, 8 for advanced personal computers, and 8, 12, 24 or 36 for graphics workstations. The color is determined by the intensity of the electron beams exciting the red, green and blue phosphors, thus this memory word must be used to modulate the intensity of these beams. There are two different alternatives to interpret the binary information in a memory word as modulation parameters for the beam intensities:

1. **True color mode** which breaks down the bits of memory word into three subfields; one for each color component. Let the number of bits used to represent red, green and blue intensities be n_r, n_g and n_b respectively, and assume that $n = n_r + n_g + n_b$ holds. The number of

producible pure red, green and blue colors are 2^{n_r}, 2^{n_g} and 2^{n_b}, and the number of all possible colors is 2^n. Since the human eye is less sensitive to blue colors than to the other two components, we usually select n_r, n_g and n_b so that: $n_r \approx n_g$ and $n_b \leq n_r$. True color mode displays distribute the available bits among the three color components in a static way, which has a disadvantage that the number of producible red colors, for instance, is still 2^{n_r} even if no other colors are to be shown on the display.

2. **Indexed color** or **pseudo color mode** which interprets the content of the frame buffer as indices into a color table called the **lookup table** or **LUT** for short. An entry in this lookup table contains three m-bit fields containing the intensities of red, green and blue components in this color. (**Gray-shade systems** have only a single field.) The typical value of m is 8. This lookup table is also a read-write memory whose content can be altered by the application program. Since the number of possible indices is 2^n, the number of simultaneously visible colors is still 2^n in indexed color mode, but these colors can be selected from a set of 2^{3m} colors. This selection is made by the proper control of the content of the lookup table. If $3m >> n$, this seems to be a significant advantage, thus the indexed color mode is very common in low-cost graphics subsystems where n is small. The lookup table must be read each time a pixel is sent to the display; that is, about every 10 nanoseconds in a high resolution display. Thus the lookup table must be made of very fast memory elements which have relatively small capacity. This makes the indexed color mode not only lose its comparative advantages when n is large, but also infeasible.

Concerning the color computation phase, the indexed color mode has another important disadvantage. When a color is generated and is being written into the frame buffer, it must be decided which color index would represent it in the most appropriate way. It generally requires a search of the lookup table and the comparison of the colors stored there with the calculated color, which is an unacceptable overhead. In special applications, such as 2D image synthesis and 3D image generation assuming very simple illumination models and only white lightsources, however, the potentially calculated colors of the primitives can be determined before the actual computation, and the actual colors can be replaced by the color indices in the

color calculation. In 2D graphics, for example, the "**visible color**" of an object is always the same as its "**own color**". (By definition the "own color" is the "visible color" when the object is lit by the sun or by an equivalent maximum intensity white lightsource.) Thus filling up the lookup table by the "own colors" of the potentially visible objects and replacing the color of the object by the index of the lookup table location where this color is stored makes it possible to use the indexed color mode. In 3D image synthesis, however, the "visible color" of an object is a complex function of the "own color" of the object, the properties of the lightsources and the camera, and the color of other objects because of the light reflection and refraction. This means that advanced 3D image generation systems usually apply true color mode, and therefore we shall only discuss true color systems in this book. Nevertheless, it must be mentioned that if the illumination models used are simplified to exclude non-diffuse reflection, refraction and shadows, and only white lightsources are allowed, then the visible color of an object will be some attenuated version of the own color. Having filled up the lookup table by the attenuated versions of the own color (the same hue and saturation but less intensity) of the potentially visible objects, and having replaced the color information by this attenuation factor in visibility computations, the color index can be calculated from the attenuation factor applying a simple linear transformation which maps [0..1] onto the range of indices corresponding to the attenuated color versions of the given object. This technique is used in *Tektronix* graphics terminals and workstations.

Even if true color mode is used — that is, the color is directly represented by the frame buffer data — the final transformation offered by the lookup tables can be useful because it can compensate for the non-linearities of the graphics monitors, known as γ-**distortion**. Since the individual color components must be compensated separately, this method requires the lookup table to be broken down into three parallelly addressable memory blocks, each of them is responsible for compensating a single color component. This method is called γ-**correction**.

As the display list for vector graphics, the frame buffer controls the intensity of the three electron beams, but now the surface of the display is scanned in the order of pixels left to right and from top to bottom in the screen. The hardware unit responsible for taking out the pixels from the frame buffer in this order, transforming them by the lookup tables and modulating the intensity of the electron beams is called the **video refresh**

controller. Since the intensity of the electron beams can be controlled by analog voltage signals, the color values represented digitally in the lookup tables or in the frame buffer must be converted to three analog signals, one for each color coordinate. This conversion is done by three digital-analog (D/A) converters. In addition to periodically refreshing the screen with the data from the frame buffer, video refresh controllers must also generate special synchronization signals for the monitors, which control the movement of the electron beam, specifying when it has to return to the left side of the screen to start scanning the next consecutive row (horizontal retrace) and when it has return to the upper left corner to start the next image (vertical retrace). In order to sustain the image on the screen, the video refresh controller must generate periodically scanning electron beams which excite the phosphors again before they start fading. The flicker-free display requires the screen to be refreshed about 60 times a second.
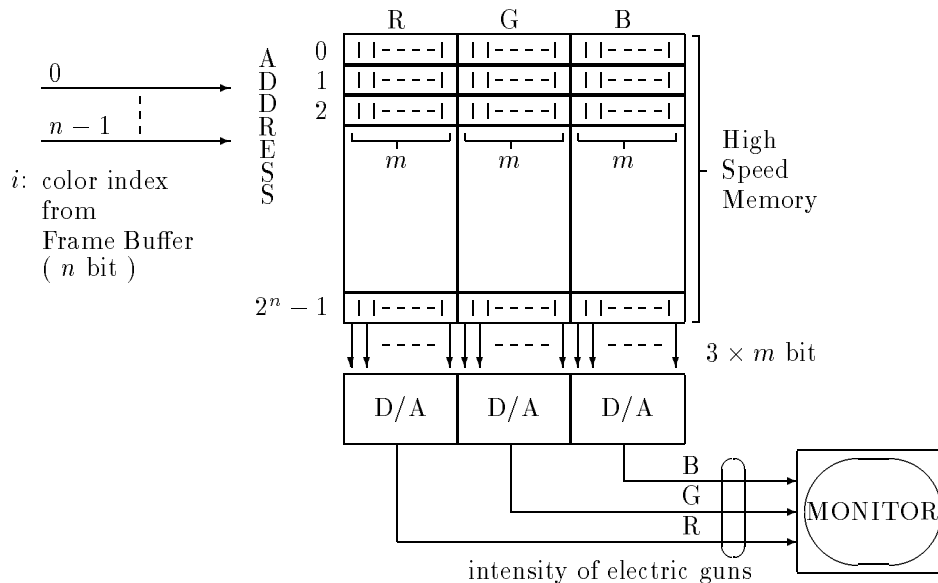
Figure 1.3: Organization of a video lookup table

The number of the pixel columns and rows is defined by the **resolution** of the graphics system. Typical resolutions are 640×480, 1024×768 for inexpensive systems and 1280×1024, 1600×1200 for advanced system. Thus an advanced workstation has over 10^6 pixels, which means that the time available to draw a single pixel, including reading it from the frame buffer and transforming it by the lookup table, is about 10 nanoseconds. This speed requirement can only be met by special hardware solutions in the video refresh controller and also by the parallel access of the frame buffer, because the required access time is much less than the cycle time of memory chips used in frame buffers. (The size of frame buffers — $1280 \times 1024 \times 24$ bits ≈ 3 Mbyte — does not allow for the application of high speed memory chips.) Fortunately, the parallelization of reading the pixel from the frame buffer is feasible because the display hardware needs the pixel data in a coherent way, that is, pixels are accessed one after the other left to right, and from top to bottom. Taking advantage of this property, when a pixel color is modulating the electron beams, the following pixels of the frame buffer row can be loaded into a shift register which in turn rolls out the pixels one-by-one at the required speed and without accessing the frame buffer. If the shift register is capable of storing N consecutive pixels, then the frequency of frame buffer accesses is decreased by N times.

A further problem arises from the fact that the frame buffer is a double access memory since the display processor writes new values into it, while the video refresh controller reads it to modulate the electron beams. Concurrent requests of the display processor and the refresh controller to read and write the frame buffer must be resolved by inhibiting one request while the other is being served. If N, the length of the shift register, is small, then the cycle time of read requests of the video refresh controller is comparable with the minimum cycle time of the memory chips, which literally leaves no time for display processor operations except during vertical and horizontal retrace. This was the reason that in early graphics systems the display processor was allowed to access the frame buffer just for a very small portion of time, which significantly decreased the drawing performance of the system. By increasing the length of the shift register, however, the time between refresh accesses can be extended, making it possible to include several drawing accesses between them. In current graphics systems, the shift registers that are integrated into the memory chips developed for frame buffer applications (called Video RAMs, or VRAMs) can hold a complete

pixel row, thus the refresh circuit of these systems needs to read the frame buffer only once in each row, letting the display processor access the frame buffer almost one hundred percent of the time.

As mentioned above, the video-refresh controller reads the content of the frame buffer periodically from left to right and from top to bottom of the screen. It uses counters to generate the consecutive pixel addresses. If the frame buffer is greater than the resolution of the screen, that is, only a portion of the pixels can be seen on the screen, the "left" and the "top" of the screen can be set dynamically by extending the counter network by "left" and "top" initialization registers. In early systems, these initialization registers were controlled to produce panning and scrolling effects on the display. Nowadays this method has less significance, since the display hardware is so fast that copying the whole frame buffer content to simulate scrolling and panning is also feasible.

There are two fundamental ways of refreshing the display: **interlaced**, and **non-interlaced**.

Interlaced refresh is used in broadcast television when the display refresh cycle is broken down into two phases, called *fields*, each lasting about 1/60 second, while a full refresh takes 1/30 second. All odd-numbered scan lines of the frame buffer are displayed in the first field, and all even-numbered lines in the second field. This method can reduce the speed requirements of the refresh logic, including frame buffer read, lookup transformation and digital-analog conversion, without significant flickering of images which consist of large homogeneous areas (as normal TV images do). However in CAD applications where, for example, one pixel wide horizontal lines possibly appear on the screen, this would cause bad flickering. TV images are continuously changing, while CAD systems allow the users to look at static images, and these static images even further emphasize the flickering effects. This is why advanced systems use non-interlaced refresh strategy exclusively, where every single refresh cycle generates all the pixels on the screen.

1.5 Image synthesis

Image synthesis is basically a transformation from model space to the color distribution of the display defined by the digital image. Its techniques
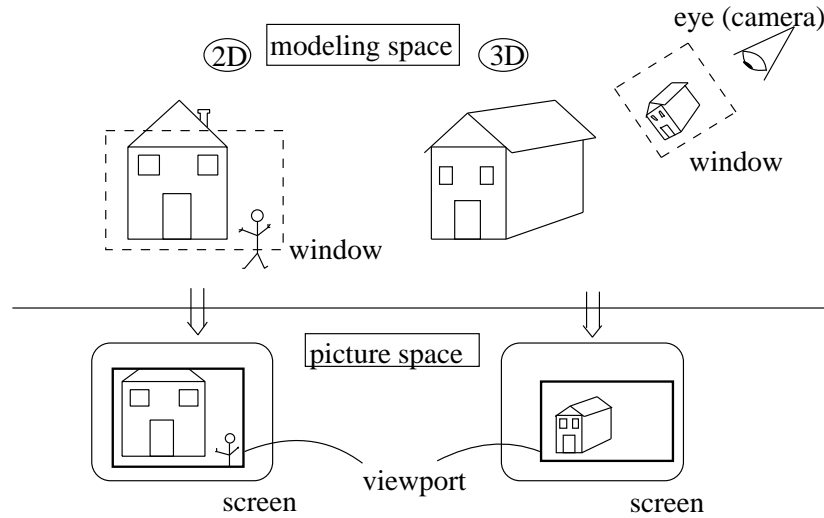
Figure 1.4: Comparison of 2D and 3D graphics

greatly depend on the space where the geometry of the internal model is represented, and we make a distinction between two- and three-dimensional graphics (2D or 3D for short) according to whether this space is two- or three-dimensional (see figure 1.4). In 2D this transformation starts by placing a rectangle, called a **2D window**, on a part of the plane of the 2D modeling space, then maps a part of the model enclosed by this rectangle to an also rectangular region of the display, called a **viewport**. In 3D graphics, the window rectangle is placed into the 3D space of the virtual world with arbitrary orientation, a **camera** or **eye** is placed behind the window, and the photo is taken by projecting the model onto the window plane having the camera as the center of projection, and ignoring those parts mapped outside the window rectangle. As in 2D, the photo is displayed in a viewport of the computer screen. Note that looking at the display only, it is not possible to decide if the picture has been generated by a two- or three-dimensional image generation method, since the resulting image is always two-dimensional. An exceptional case is the holographic display, but this topic is not covered in this book. On the other hand, a technique, called **stereovision**, is the proper combination of two normal images for the two eyes to emphasize the 3D illusion.

In both 2D and 3D graphics, the transformation from the model to the color distribution of the screen involves the following characteristic steps:

- **Object-primitive decomposition:** As has been emphasized, the internal world stores information in a natural way from the point of view of the modeling process so as to allow for easy modification and not to restrict the analysis methods used. In an architectural program, for example, a model of a house might contain building blocks such as a door, a chimney, a room etc. A general purpose image synthesis program, however, deals with primitives appropriate to its own internal algorithms such as line segments, polygons, parametric surfaces etc., and it cannot be expected to work directly on objects like doors, chimneys etc. This means that the very first step of the image generation process must be the decomposition of real objects used for modeling into primitives suitable for the image synthesis algorithms.

- **Modeling transformation:** Objects are defined in a variety of local coordinate systems: the nature of the system will depend on the nature of the object. Thus, to consider their relative position and orientation they have to be transferred to the global coordinate system by a transformation associated with them.

- **World-screen transformation:** Once the modeling transformation stage has been completed, the geometry of the model will be available in the global world coordinate system. However, the generated image is required in a coordinate system of the screen since eventually the color distribution of the screen has to be determined. This requires another geometric transformation which maps the 2D window onto the viewport in the case of 2D, but also involving projection in the case of 3D, since the dimension of the representation has to be reduced from three to two.

- **Clipping:** Given the intuitive process of taking photos in 2D and 3D graphics, it is obvious that the photo will only reproduce those portions of the model which lie inside the 2D window, or in the infinite pyramid defined by the camera as the apex, and the sides of the 3D window. The 3D infinite pyramid is usually limited to a finite frustum of pyramid to avoid overflows, thus forming a **front clipping plane**

and a **back clipping plane** parallel to the window. The process of removing those invisible parts that fall outside either the 2D window or the viewing frustum of pyramid is called clipping. It can either be carried out before the world-screen transformation, or else during the last step by inhibiting the modification of those screen regions which are outside the viewport. This latter process is called **scissoring**.

- **Visibility computations:** Window-screen transformations may project several objects onto the same point on the screen if they either overlap or if they are located behind each other. It should be decided which object's color is to be used to set the color of the display. In 3D models the object selected should be the object which hides others from the camera; i.e. of all those objects that project onto the same point in the window the one which is closest to the camera. In 2D no geometry information can be relied on to resolve the visibility problem but instead an extra parameter, called **priority**, is used to select which object will be visible. In both 2D and 3D the visibility computation is basically a sorting problem based on the distance from the eye in 3D, and on the priority in 2D.

- **Shading:** Having decided which object will be visible at a point on the display its color has to be calculated. This color calculation step is called shading. In 2D this step poses no problem because the object's **own color** should be used. An object's "own color" can be defined as the perceived color when only the sun, or an equivalent lightsource having the same energy distribution, illuminates the object. In 3D, however, the perceived color of an object is a complex function of the object's own color, the parameters of the lightsources and the reflections and refractions of the light. Theoretically the models and laws of geometric and physical optics can be relied on to solve this problem, but this would demand lengthy computational process. Thus approximations of the physical models are used instead. The degree of the approximation also defines the level of compromise in image generation speed and quality.

Comparing the tasks required by 2D and 3D image generation we can see that 3D graphics is more complex at every single stage, but the difference really becomes significant in visibility computations and especially in

shading. That is why so much of this book will be devoted to these two topics.

Image generation starts with the manipulation of objects in the virtual world model, later comes the transformation, clipping etc. of graphics primitives, and finally, in raster graphics systems, it deals with pixels whose constant color will approximate the continuous image.

Algorithms playing a part in image generation can thus be classified according to the basic type of data handled by them:

1. **Model decomposition algorithms** decompose the application oriented model into **graphics primitives** suitable for use by the subsequent algorithms.

2. **Geometric manipulations** include transformations and clipping, and may also include visibility and shading calculations. They work on graphics primitives independently of the resolution of the raster storage. By arbitrary definition all algorithms belong to this category which are independent of both the application objects and the raster resolution.

3. **Scan conversion algorithms** convert the graphics primitives into pixel representations, that is, they find those pixels, and may determine the colors of those pixels which approximate the given primitive.

4. **Pixel manipulations** deal with individual pixels and eventually write them to the raster memory (also called *frame buffer memory*).

1.6   Modeling and world representation

From the point of view of image synthesis, **modeling** is a necessary preliminary phase that generates a database called **virtual world representation**. Image synthesis operates on this database when taking "synthetic photos" of it. From the point of view of modeling on the other hand, image synthesis is only one possible analysis procedure that can be performed on the database produced. In industrial computer-aided design and manufacturing (CAD/CAM), for example, geometric models of products can be used to calculate their volume, mass, center of mass, or to generate a

sequence of commands for a numerically controlled (NC) machine in order to produce the desired form from real material, etc. Thus image synthesis cannot be treated separately from modeling, but rather its actual operation highly depends on the way of representing the **scene** (collection of objects) to be rendered. (This can be noticed when one meets such sentences in the description of rendering algorithms: "Assume that the objects are described by their bounding polygons ..." or "If the objects are represented by means of set operations performed on simple geometric forms, then the following can be done ...", etc.)

Image synthesis requires the following two sorts of information to be included in the virtual world representation:

1. *Geometric information.* No computer program can render an object without information about its shape. The shape of the objects must be represented by numbers in the computer memory. The field of **geometric modeling** (or solid modeling, shape modeling) draws on many branches of mathematics (geometry, computer science, algebra). It is a complicated subject in its own right. There is not sufficient space in this book to fully acquaint the reader with this field, only some basic notions are surveyed in this section.

2. *Material properties.* The image depends not only on the geometry of the scene but also on those properties of the objects which influence the interaction of the light between them and the lightsources and the camera. Modeling these properties implies the characterization of the object surfaces and interiors from an optical point of view and the modeling of light itself. These aspects of image synthesis are explained in chapter 3 (on physical modeling of 3D image synthesis).

1.6.1   Geometric modeling

The terminology proposed by Requicha [Req80] still seems to be general enough to describe and characterize geometric modeling schemes and systems. This will be used in this brief survey.

Geometric and graphics algorithms manipulate *data structures* which (may) *represent* physical solids. Let D be some domain of data structures. We say that a data structure $d \in D$ represents a physical solid if there is a

mapping $m\colon D \to E^3$ (E^3 is the 3D Euclidean space), for which $m(d)$ *models* a physical solid. A subset of E^3 models a physical solid if its shape can be produced from some real material. Subsets of E^3 which model physical solids are called **abstract solids**. The class of abstract solids is very small compared to the class of all subsets of E^3.

Usually the following properties are required of abstract solids and representation methods:

1. *Homogeneous 3-dimensionality.* The solid must have an interior of positive volume and must not have isolated or "dangling" (lower dimensional) portions.

2. *Finiteness.* It must occupy a finite portion of space.

3. *Closure under certain Boolean operations.* Operations that model working on the solid (adding or removing material) must produce other abstract solids.

4. *Finite describability.* The data structure describing an abstract solid must have a finite extent in order to fit into the computer's memory.

5. *Boundary determinism.* The boundary of the solid must unambiguously determine which points of E^3 belong to the solid.

6. (*Realizability.* The shape of the solid should be suitable for production from real material. Note that this property is not required for producing virtual reality.)

The mathematical implications of the above properties are the following. Property 1 requires the abstract solid to belong to the class of **regular sets**. In order to define regular sets in a self-contained manner, some standard notions of set theory (or set theoretical topology) must be recalled here [KM76], [Men75], [Sim63]. A neighborhood of a point p, denoted by $N(p)$, can be any set for which $p \in N(p)$. For any set S, its complement (cS), interior (iS), closure (kS) and boundary (bS) are defined using the notion of neighborhood:

$$
\begin{aligned}
cS &= \left\{ p \mid p \notin S \right\}, \\
iS &= \left\{ p \mid \exists N(p)\colon N(p) \subset S \right\}, \\
kS &= \left\{ p \mid \forall N(p)\colon \exists q \in N(p)\colon q \in S \right\}, \\
bS &= \left\{ p \mid p \in kS \text{ and } p \in kcS \right\}.
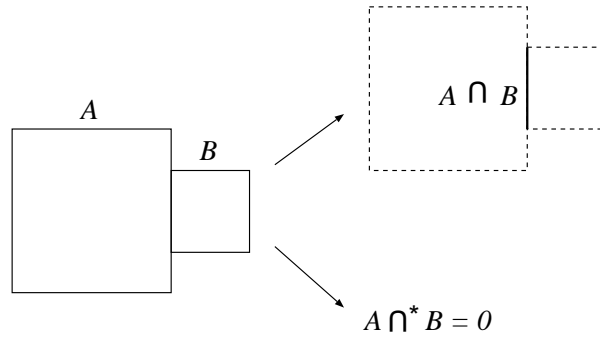\end{aligned}
\tag{1.1}
$$

Figure 1.5: An example when regularized set operation ($\cap^*$) is necessary

Then a set S is defined as regular, if:

$$S = kiS. \tag{1.2}$$

Property 2 implies that the solid is *bounded*, that is, it can be enclosed by a sphere of finite volume. Property 3 requires the introduction of the **regularized set operations**. They are derived from the ordinary set operations ($\cup, \cap, \backslash$ and the complement c) by abandoning non-3D ("dangling") portions of the resulting set. Consider, for example, the situation sketched in figure 1.5, where two cubes, A and B, share a common face, and their intersection $A \cap B$ is taken. If $\cap$ is the ordinary set-theoretical intersection operation, then the result is a dangling face which cannot correspond to a real 3D object. The regularized intersection operation ($\cap^*$) should give the empty set in this case. Generally if $\circ$ is a binary set operation ($\cup, \cap$ or $\backslash$) in the usual sense, then its regularized version $\circ^*$ is defined as:

$$A \circ^* B = ki\left(A \circ B\right). \tag{1.3}$$

The unary complementing operation can be regularized in a similar way:

$$c^*A = ki\left(cA\right). \tag{1.4}$$

The regular subsets of E^3 together with the regularized set operations form a Boolean algebra [Req80]. Regularized set operations are of great importance in some representation schemes (see CSG schemes in subsection

1.6.2). Property 4 implies that the shape of the solids is defined by some formula (or a finite system of formulae) F: those and only those points of E^3 which satisfy F belong to the solid. Property 5 has importance when the solid is defined by its boundary because this boundary must be valid (see B-rep schemes in subsection 1.6.2). Property 6 requires that the abstract solid is a *semianalytic* set. It poses constraints on the formula F. A function $f\colon E^3 \to R$ is said to be analytic (in a domain) if $f(x, y, z)$ can be expanded in a convergent power series about each point (of the domain). A subset of analytic functions is the set of *algebraic* functions which are polynomials (of finite degree) in the coordinates x, y, z. A set is semianalytic (semialgebraic) if it can be expressed as a finite Boolean combination (using the set operations $\cup, \cap, \backslash$ and c or their regularized version) of sets of the form:

$$S_i = \{x, y, z\colon f_i(x, y, z) \le 0\}, \tag{1.5}$$

where the functions f_i are analytic (algebraic). In most practical cases, semialgebraic sets give enough freedom in shape design.

The summary of this subsection is that suitable models for solids are subsets of E^3 that are *bounded, closed, regular* and *semianalytic (semialgebraic)*. Such sets are called **r-sets**.

1.6.2   Example representation schemes

A **representation scheme** is the correspondence between a data structure and the abstract solid it represents. A given representation scheme is also a given method for establishing this connection. Although there are several such methods used in practical *geometric modeling systems* only the two most important are surveyed here.

Boundary representations (B-rep)

The most straightforward way of representing an object is describing its boundary. Students if asked about how they would represent a geometric form by a computer usually choose this way.

A solid can be really well represented by describing its boundary. The boundary of the solid is usually segmented into a finite number of bounded subsets called **face**s or **patch**es and each face is represented separately. In the case of polyhedra, for example, the faces are planar polygons and hence
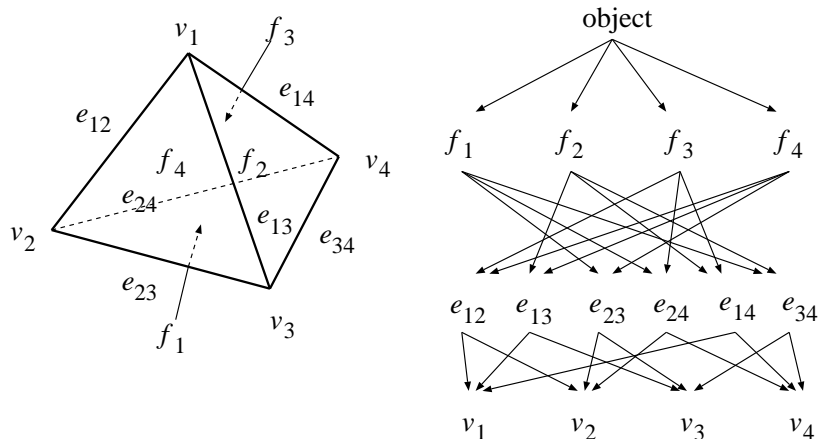
Figure 1.6: B-rep scheme for a tetrahedron

can be represented by their bounding edges and vertices. Furthermore, since the edges are straight line segments, they can be represented by their bounding vertices. Figure 1.6 shows a tetrahedron and a possible B-rep scheme. The representation is a directed graph containing object, face, edge and vertex nodes. Note that although only the lowest level nodes (the vertex nodes) carry geometric information and the others contain only "pure topological' information in this case, it is not always true, since in the general case, when the shape of the solid can be arbitrarily curved (sculptured), the face and edge nodes must also contain shape information.

The *validity* of a B-rep scheme (cf. property 5 in the beginning of this subsection) requires the scheme to meet certain conditions. We distinguish two types of validity conditions:

1. *combinatorial (topological)* conditions: (1) each edge must have precisely two vertices; (2) each edge must belong to an even number of faces;

2. *metric (geometric)* conditions: (1) each vertex must represent a distinct point of E^3; (2) edges must either be disjoint or intersect at a common vertex; (3) faces must either be disjoint or intersect at a common edge or vertex.

These conditions do not exclude the representation of so-called **non-manifold** objects. Let a solid be denoted by S and its boundary by ∂S. The solid S is said to be **manifold** if each of its boundary points $p \in \partial S$ has a neighborhood $N(p)$ (with positive volume) for which the set $N(p) \cap \partial S$ (the neighborhood of p on the boundary) is *homeomorphic* to a disk. (Two sets are homeomorphic if there exists a continuous one-to-one mapping which transforms one into the other.) The union of two cubes sharing a common edge is a typical example of a non-manifold object, since each point on the common edge becomes a "non-manifold point". If only two faces are allowed to meet at each edge (cf. combinatorial condition (2) above), then the scheme is able to represent only manifold objects. The **winged edge** data structure, introduced by Baumgart [Bau72], is a boundary representation scheme which is capable of representing manifold objects and inherently supports the automatic examination of the above listed combinatorial validity conditions. The same data structure is known as **doubly connected edge list (DCEL)** in the context of computational geometry, and is described in section 6.7.

Constructive solid geometry (CSG) representations

Constructive solid geometry (CSG) includes a family of schemes that represent solids as Boolean constructions or combinations of solid components via the regularized set operations ($\cup^*, \cap^*, \backslash^*, c^*$). CSG representations are binary trees. See the example shown in figure 1.7. Internal (nonterminal) nodes represent set operations and leaf (terminal) nodes represent subsets (r-sets) of E^3. Leaf objects are also known as *primitives*. They are usually simple bounded geometric forms such as blocks, spheres, cylinders, cones or unbounded halfspaces (defined by formulae such as $f(x, y, z) \leq 0$). A more general form of the CSG-tree is when the nonterminal nodes represent either set operations or rigid motions (orientation and translation transformations) and the terminal nodes represent either primitives or the parameters of rigid motions.

The *validity* of CSG-trees poses a smaller problem than that of B-rep schemes: if the primitives are r-sets (that is general unbounded halfspaces are not allowed), for example, then the tree always represents a valid solid.

Figure 1.7: A CSG scheme

Note that a B-rep model is usually closer to being ready for image synthesis than a CSG representation since primarily surfaces can be drawn and not volumes. Transforming a CSG model into a (approximate) B-rep scheme will be discussed in section 4.2.2 in the context of model decomposition.

Chapter 2

ALGORITHMICS FOR IMAGE GENERATION

Before going into the details of various image synthesis algorithms, it is worth considering their general aspects, and establishing a basis for their comparison in terms of efficiency, ease of realization, image quality etc., because it is not possible to understand the specific steps, and evaluate the merits or drawbacks of different approaches without keeping in mind the general objectives. This chapter is devoted to the examination of algorithms in general, what has been called algorithmics after the excellent book of D. Harel [Har87].

Recall that a complete image generation consists of model decomposition, geometric manipulation, scan conversion and pixel manipulation algorithms. The resulting picture is their collective product, each of them is responsible for the image quality and for the efficiency of the generation.

Graphics algorithms can be compared, or evaluated by considering the reality of the generated image, that is how well it provides the illusion of photos of the real world. Although this criterion seems rather subjective, we can accept that the more accurately the model used approximates the laws of nature and the human perception, the more realistic the image which can be expected. The applied laws of nature fall into the category of geometry and optics. Geometrical accuracy regards how well the algorithm sustains the original geometry of the model, as, for example, the image of a sphere is expected to be worse if it is approximated by a polygon mesh during synthesis than if it were treated as a mathematical object defined by the

equation of the sphere. Physical, or optical accuracy, on the other hand, is based on the degree of approximations of the laws of geometric and physical optics. The quality of the image will be poorer, for example, if the reflection of the light of indirect lightsources is ignored, than if the laws of reflection and refraction of geometrical optics were correctly built into the algorithm.

Image synthesis algorithms are also expected to be fast and efficient and to fit into the memory constraints. In real-time animation the time allowed to generate a complete image is less than 100 msec to provide the illusion of continuous motion. In **interactive systems** this requirement is not much less severe if the system has to allow the user to control the camera position by an interactive device. At the other extreme end, high quality pictures may require hours or even days on high-performance computers, thus only 20–30 % decrease of computational time would save a great amount of cost and time for the user. To describe the time and storage requirements of an algorithm independently of the computer platform, **complexity measures** have been proposed by a relatively new field of science, called theory of computation. Complexity measures express the rate of the increase of the required time and space as the size of the problem grows by providing upper and lower limits or asymptotic behavior. The problem size is usually characterized by the number of the most important data elements involved in the description and in the solution of the problem.

Complexity measures are good at estimating the applicability of an algorithm as the size of the problem becomes really big, but they cannot provide characteristic measures for a small or medium size problem, because they lack the information of the time unit of the computations. An algorithm having $const_1 \cdot n^2$ computational time requirement in terms of problem size n, denoted usually by $\Theta(n^2)$, can be better than an algorithm of $const_2 \cdot n$, or $\Theta(n)$, if $const_1 << const_2$ and n is small. Consequently, the time required for the computation of a "unit size problem" is also critical especially when the total time is limited and the allowed size of the problem domain is determined from the overall time requirement. The unit calculation time can be reduced by the application of more powerful computers. The power of general purpose processors, however, cannot meet the requirements of constantly increasing expectations of the graphics community. A real-time animation system, for example, has to generate at least 15 images per second to provide the illusion of continuous motion. Suppose that the number of pixels on the screen is about 10^6 (advanced systems usually have

1280×1024 resolution). The maximum average time taken to manipulate a single pixel (t_{pixel}), which might include visibility and shading calculations, cannot exceed the following limit:

$$t_{\mathrm{pixel}} < \frac{1}{15 \cdot 10^6} \approx 66 \text{ nsec.} \qquad (2.1)$$

Since this value is less than a single commercial memory read or write cycle, processors which execute programs by reading the instructions and the data from memories are far too slow for this task, thus special solutions are needed, including:

1. **Parallelization** meaning the application of many computing units running parallelly, and allocating the computational burden between the parallel processors. Parallelization can be carried out on the level of processors, resulting in multiprocessor systems, or inside the processor, which leads to special graphics chips capable of computing several pixels parallelly and handling tasks such as instruction fetch, execution and data transfer simultaneously.

2. **Hardware realization** meaning the design of a special digital network instead of the application of general purpose processors with information about the algorithm contained by the architecture of the hardware, not by a separate software component as in general purpose systems.

The study of the hardware implementation of algorithms is important not only for hardware engineers but for everyone involved in computer graphics, since the requirements of an effective software realization are quite similar to those indispensable for hardware translation. It means that a transformed algorithm ready for hardware realization can run faster on a general purpose computer than a naive implementation of the mathematical formulae.

2.1 Complexity of algorithms

Two complexity measures are commonly used to evaluate the effectiveness of algorithms: the **time** it spends on solving the problem (calculating the result) and the size of **storage** (memory) it uses to store its own temporary

data in order to accelerate the calculations. Of course, both the time and storage spent on solving a given problem depend on the one hand on the *nature of the problem* and on the other hand on the *amount* of the input data. If, for example, the problem is to find the greatest number in a list of numbers, then the size of the input data is obviously the length of the list, say n. In this case, the time complexity is usually given as a function of n, say $T(n)$, and similarly the storage complexity is also a function of n, say $S(n)$. If no preliminary information is available about the list (whether the numbers in it are ordered or not, etc.), then the algorithm must examine each number in order to decide which is the greatest. It follows from this that the time complexity of *any* algorithm finding the greatest of n numbers is *at least proportional to n*. It is expressed by the following notation:

$$T(n) = \Omega(n). \tag{2.2}$$

A rigorous definition of this and the other usual complexity notations can be found in the next subsection. Note that such statements can be made without having any algorithm for solving the given problem, thus such **lower bounds** are related rather to the problems themselves than to the concrete algorithms. Let us then examine an obvious algorithm for solving the maximum-search problem (the input list is denoted by $k_1, \ldots, k_n$):

> **FindMaximum**$(k_1, \ldots, k_n)$
> $M = k_1;$ // *M: the greatest found so far*
>   **for** $i = 2$ **to** n **do**
>   **if** $k_i > M$ **then**
> $M = k_i;$
>   **endif**
>   **endfor**
>   **return** $M;$
> **end**

Let the time required by the assignment operator $(=)$ be denoted by $T_=$, the time required to perform the comparison $(k_i > M)$ by $T_>$ and the time needed to prepare for a cycle by T_{loop} (the time of an addition and a comparison).

The time T spent by the above algorithm can then be written as:

$$T = T_= + (n-1) \cdot T_> + m \cdot T_= + (n-1) \cdot T_{\mathrm{loop}} \quad (m \le n-1), \tag{2.3}$$

where m is number of situations when the variable M must be updated. The value of m can be $n-1$ in the **worst case** (that is when the numbers in the input list are in ascending order). Thus:

$$T \leq T_= + (n-1) \cdot (T_> + T_= + T_{\text{loop}}).$$ (2.4)

The conclusion is that the time spent by the algorithm is *at most proportional* to n. This is expressed by the following notation:

$$T(n) = O(n).$$ (2.5)

This, in fact, gives an **upper bound** on the complexity of the maximum-searching problem itself: it states that there exists an algorithm that can solve it in time proportional to n. The lower bound ($T(n) = \Omega(n)$) and the worst-case time complexity of the proposed algorithm ($T(n) = O(n)$) coincide in this case. Hence we say that the algorithm has an **optimal** (worst-case optimal) time complexity. The storage requirement of the algorithm is only one memory location that stores M, hence the storage complexity is independent of n, that is constant:

$$S(n) = O(1).$$ (2.6)

2.1.1 Complexity notations

In time complexity analysis usually not all operations are counted but rather only those ones that correspond to a *representative* set of operations called **key operations**, such as comparisons or assignments in the previous example. (The key operations should always be chosen carefully. In the case of matrix-matrix multiplication, as another example, the key operations are multiplications and additions.) The number of the actually performed key operations is expressed as a function of the input size. In doing so, one must ensure that the number (execution time) of the unaccounted-for operations is at most proportional to that of the key operations so that the running time of the algorithm is within a constant factor of the estimated time. In storage complexity analysis, the maximum amount of storage ever required during the execution of the algorithm is measured, also expressed as a function of the input size. However, instead of expressing these functions exactly, rather their **asymptotic behavior** is analyzed, that is when

the input size approaches infinity, and expressed by the following special notations.

The notations must be able to express both that the estimations are valid only within a constant factor and that they reflect the asymptotic behavior of the functions. The so-called "big-O" and related notations were originally suggested by Knuth [Knu76] and have since become standard complexity notations [PS85].

Let $f, g : N \mapsto R$ be two real-valued functions over the integer numbers. The notation

$$f = O(g) \tag{2.7}$$

denotes that we can find $c > 0$ and $n_0 \in N$ so that $f(n) \leq c \cdot g(n)$ if $n > n_0$, that is, the function f *grows at most at the rate* of g in asymptotic sense. In other words, g is an **upper bound** of f. For example, $n^2 + 3n + 1 = O(n^2) = O(n^3) = \ldots$ but $n^2 + 3n + 1 \neq O(n)$. The notation

$$f = \Omega(g) \tag{2.8}$$

denotes that we can find $c > 0$ and $n_0 \in N$ so that $f(n) \geq c \cdot g(n)$ if $n > n_0$, that is, f grows *at least at the rate* of g. In other words, g is a **lower bound** of f. For example, $n^2 + 3n + 1 = \Omega(n^2) = \Omega(n)$. Note that $f = \Omega(g)$ is equivalent with $g = O(f)$. Finally, the notation

$$f = \Theta(g) \tag{2.9}$$

denotes that we can find $c_1 > 0, c_2 > 0$ and $n_0 \in N$ so that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ if $n > n_0$, that is, f *grows exactly at the rate* of g. Note that $f = \Theta(g)$ is equivalent with $f = O(g)$ and $f = \Omega(g)$ at the same time.

An interesting property of complexity classification is that it is *maximum emphasizing* with respect to weighted sums of functions, in the following way. Let the function $H(n)$ be defined as the positively weighted sum of two functions that belong to different classes:

$$H(n) = a \cdot F(n) + b \cdot G(n) \quad (a, b > 0) \tag{2.10}$$

where

$$F(n) = O(f(n)), \quad G(n) = O(g(n)), \quad g(n) \neq O(f(n)), \tag{2.11}$$

that is, $G(n)$ belongs to a higher class than $F(n)$. Then their combination, $H(n)$, belongs to the higher class:

$$H(n) = O(g(n)), \quad H(n) \neq O(f(n)). \tag{2.12}$$

Similar statements can be made about Ω and Θ.

The main advantage of the notations introduced in this section is that statements can be formulated about the complexity of algorithms in a hardware-independent way.

2.1.2 Complexity of graphics algorithms

Having introduced the "big-O" the effectiveness of an algorithm can be formalized. An alternative interpretation of the notation is that $O(f(n))$ denotes the class of all functions that grow not faster than f as $n \to \infty$. It defines a nested sequence of function classes:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(a^n) \tag{2.13}$$

where the basis of the logarithm can be any number greater than one, since the change of the basis can be compensated by a constant factor. Note, however, that this is not true for the basis a of the power ($a > 1$).

Let the time complexity of an algorithm be $T(n)$. Then the smaller the smallest function class containing $T(n)$ is, the faster the algorithm is. The same is true for storage complexity (although this statement would require more preparation, it would be so similar to that of time complexity that it is left for the reader).

When analyzing an algorithm, the goal is always to find the *smallest upper bound*, but it is not always possible. When constructing an algorithm, the goal is always to reach the *tightest known lower bound* (that is to construct an optimal algorithm), but it is neither always possible.

In **algorithm theory**, an algorithm is "good" if $T(n) = O(n^k)$ for some finite k. These are called *polynomial* algorithms because their running time is at most proportional to a polynomial of the input size. A given computational problem is considered as *practically tractable* if a polynomial algorithm exists that computes it. The *practically non-tractable* problems are those for which no polynomial algorithm exists. Of course, these problems

can also be solved computationally, but the running time of the possible algorithms is at least $O(a^n)$, that is *exponentially* grows with the input size.

In computer graphics or generally in CAD, where in many cases real-time answers are expected by the user (interactive dialogs), the borderline between "good" and "bad" algorithms is drawn much lower. An algorithm with a time complexity of $O(n^{17})$, for example, can hardly be imagined as a part of a CAD system, since just duplicating the input size would cause the processing to require 2^{17} (more than 100,000) times the original time to perform the same task on the bigger input. Although there is no commonly accepted standard for distinguishing between acceptable and non-acceptable algorithms, the authors' opinion is that the practical borderline is somewhere about $O(n^2)$.

A further important question arises when estimating the effectiveness of graphics or generally, geometric algorithms: what should be considered as the **input size**? If, for example, triangles (polygons) are to be transformed from one coordinate system into another one, then the total number of vertices is a proper measure of the input size, since these shapes can be transformed by transforming its vertices. If n is the number of vertices then the complexity of the transformation is $O(n)$ since the vertices can be transformed independently. If n is the number of triangles then the complexity is the same since each triangle has the same number of (three) vertices. Generally the input size (problem size) is the number of (usually simple) similar objects to be processed.

If the triangles must be drawn onto the screen, then the more pixels they cover the more time is required to paint each triangle. In this case, the size of the input is better characterized by the number of pixels covered than by the total number of vertices, although the number of pixels covered is related rather to the **output size**. If the number of triangles is n and they cover p pixels altogether (counting overlappings) then the time complexity of drawing them onto the screen is $O(n + p)$ since each triangle must be first transformed (and projected) and then painted. If the running time of an algorithm depends not only on the size of the input but also on the size of the output, then it is called an **output sensitive algorithm**.

2.1.3 Average-case complexity

Sometimes the worst-case time and storage complexity of an algorithm is very bad, although the situations responsible for the worst cases occur very rarely compared to all the possible situations. In such cases, an **average-case** estimation can give a better characterization than the standard worst-case analysis. A certain probability distribution of the input data is assumed and then the expected time complexity is calculated. Average-case analysis is not as commonly used as worst-case analysis because of the following reasons:

- The worst-case complexity and the average-case complexity for any reasonable distribution of input data coincide in many cases (just as for the maximum-search algorithm outlined above).

- The probability distribution of the input data is usually not known. It makes the result of the analysis questionable.

- The calculation of the expected complexity involves hard mathematics, mainly integral calculus. Thus average-case analysis is usually not easy to perform.

Although one must accept the above arguments (especially the second one), the following argument puts average-case analysis into new light.

Consider the problem of computing the convex hull of a set of n distinct points in the plane. (The convex hull is the smallest convex set containing all the points. It is a convex polygon in the planar case with its vertices coming from the point set.) It is known [Dév93] that the lower bound of the time complexity of *any* algorithm that solves this problem is $\Omega(n \log n)$. Although there are many algorithms computing the convex hull in the optimal $O(n \log n)$ time (see Graham's pioneer work [Gra72], for example), let us now consider another algorithm having a worse worst-case but an optimal average-case time complexity. The algorithm is due to Jarvis [Jar73] and is known as "gift wrapping". Let the input points be denoted by:

$$p_1, \ldots, p_n. \tag{2.14}$$

The algorithm first searches for an extremal point in a given direction. This point can be that with the smallest x-coordinate, for example. Let

it be denoted by p_{i_1}. This point is definitely a vertex of the convex hull. Then a direction vector $\vec{d}$ is set so that the line having this direction and going through p_{i_1} is a supporting line of the convex hull, that is, it does not intersect its interior. With the above choice for p_{i_1}, the direction of $\vec{d}$ can be the direction pointing vertically downwards. The next vertex of the convex hull, p_{i_2}, can then be found by searching for that point $p \in \{p_1, \ldots, p_n\} \setminus p_{i_1}$ for which the angle between the direction of $\vec{d}$ and the direction of $\vec{p_{i_1}p}$ is minimal. The further vertices can be found in a very similar way by first setting $\vec{d}$ to $\vec{p_{i_1}p_{i_2}}$ and p_{i_2} playing the role of p_{i_1}, etc. The search continues until the first vertex, p_{i_1}, is discovered again. The output of the algorithm is a sequence of points:

$$p_{i_1}, \ldots, p_{i_m} \tag{2.15}$$

where $m \leq n$ is the size of the convex hull. The time complexity of the algorithm is:

$$T(n) = O(mn) \tag{2.16}$$

since finding the smallest "left bend" takes $O(n)$ time in each of the m main steps. Note that the algorithm is *output sensitive*. The maximal value of m is n, hence the worst-case time complexity of the algorithm is $O(n^2)$.

Let us now recall an early result in geometric probability, due to Rényi and Sulanke [RS63] (also in [GS88]): the average size of the convex hull of n random points independently and **uniformly distributed** in a triangle is:

$$E[m] = O(\log n). \tag{2.17}$$

This implies that the average-case time complexity of the "gift wrapping" algorithm is:

$$E[T(n)] = O(n \log n). \tag{2.18}$$

The situation is very interesting: the average-case complexity belongs to a lower function *class* than the worst-case complexity; the difference between the two cases cannot be expressed by a constant factor but rather it grows infinitely as n approaches infinity! What does this mean?

The n input objects of the algorithm can be considered as a *point* of a multi-dimensional *configuration space*, say K_n. In the case of the convex hull problem, for example, $K_n = R^{2n}$, since each planar point can be defined by two coordinates. In average-case analysis, each point of the configuration space is given a non-zero probability (density). Since there is no reason

for giving different probability to different points, a *uniform* distribution is assumed, that is, each point of K_n has the same probability (density). Of course, the configuration space K_n must be bounded in order to be able to give non-zero probability to the points. This is why Rényi and Sulanke chose a triangle, say T, to contain the points and K_n was $T \times T \times \ldots \times T = T^n$ in that case. Let the time spent by the algorithm on processing a given configuration $K \in K_n$ be denoted by $\tau(K)$. Then, because of uniform distribution, the expected time complexity can be calculated as:

$$\mathrm{E}\left[T(n)\right] = \int\limits_{K_n} \frac{1}{|K_n|} \tau(K) dK, \qquad (2.19)$$

where $|\cdot|$ denotes volume. The asymptotic behavior of $\mathrm{E}[T(n)]$ (as $n \to \infty$) characterizes the algorithm in the expected case. It belongs to a function class, say $O(f(n))$. Let the smallest function class containing the worst-case time complexity $T(n)$ be denoted by $O(g(n))$. The interesting situation is when $O(f(n)) \neq O(g(n))$, as in the case of "gift wrapping".

One more observation is worth mentioning here. It is in connection with the maximum-emphasizing property of the "big-O" classification, which was shown earlier (section 2.1.1). The integral 2.19 is the continuous analogue of a weighted sum, where the infinitesimal probability $dK/|K_n|$ plays the role of the weights a, b in equations 2.10–2.12. How can it then happen that, although the weight is the same everywhere in K_n (analogous to $a = b$), the result function belongs to a lower class than the worst-case function which is inevitably present in the summation? The answer is that the ratio of the situations "responsible for the worst-case" complexity and all the possible situations *tends to zero* as n grows to infinity. (A more rigorous discussion is to appear in [Már94].)

2.2   Parallelization of algorithms

Parallelization is the application of several computing units running parallelly to increase the overall computing speed by distributing the computational burden between the parallel processors.

As we have seen, image synthesis means the generation of pixel colors approximating an image of the graphics primitives from a given point of view.

More precisely, the input of this image generation is a collection of graphics **primitives** which are put through a series of **operations** identified as transformations, clipping, visibility calculations, shading, pixel manipulations and frame buffer access, and produce the **pixel data** stored in the frame buffer as output.
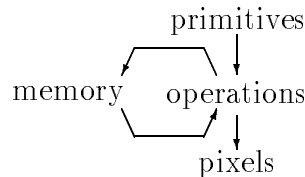

Figure 2.1: Key concepts of image generation

The key concepts of image synthesis (figure 2.1) — primitives, operations and pixels — form a simple structure which can make us think that operations represent a machine into which the primitives are fed one after the other and which generates the pixels, but this is not necessarily true. The final image depends not only on the individual primitives but also on their relationships used in visibility calculations and in shading. Thus, when a primitive is processed the "machine" of operations should be aware of the necessary properties of all other primitives to decide, for instance, whether this primitive is visible in a given pixel. This problem can be solved by two different approaches:

1. When some information is needed about a primitive it is input again into the machine of operations.

2. The image generation "machine" builds up an internal memory about the already processed primitives and their relationships, and uses this memory to answer questions referring to more than one primitives.

Although the second method requires redundant storage of information and therefore has additional memory requirements, it has several significant advantages over the first method. It does not require the model decomposition phase to run more times than needed, nor does it generate random order query requests to the model database. The records of the database can be

accessed once in their natural (most effective) order. The internal memory of the image synthesis machine can apply clever data structures optimized for its own algorithms, which makes its access much faster than the access of modeling database. When it comes to parallel implementation, these advantages become essential, thus only the second approach is worth considering as a possible candidate for parallelization. This decision, in fact, adds a fourth component to our key concepts, namely the internal **memory of primitive properties** (figure 2.1). The actual meaning of the "primitive properties" will be a function of the algorithm used in image synthesis.

When we think about realizing these algorithms by parallel hardware, the algorithms themselves must also be made suitable for parallel execution, which requires the decomposition of the original concept. This decomposition can either be accomplished functionally — that is, the algorithm is broken down into operations which can be executed parallelly — or be done in data space when the algorithm is broken down into similar parallel branches working with a smaller amount of data. Data decomposition can be further classified into input data decomposition where a parallel branch deals with only a portion of the input primitives, and output data decomposition where a parallel branch is responsible for producing the color of only a portion of the pixels. We might consider the parallelization of the memory of primitive properties as well, but that is not feasible because this memory is primarily responsible for storing information needed to resolve the dependence of primitives in visibility and shading calculations. Even if visibility, for instance, is calculated by several computing units, all of them need this information, thus it cannot be broken down into several independent parts. If separation is needed, then this has to be done by using redundant storage where each separate unit contains nearly the same information. Thus, the three basic approaches of making image synthesis algorithms parallel are:

1. **Functional decomposition** or **operation based parallelization** which allocates a different hardware unit for the different phases of the image synthesis. Since a primitive must go through every single phase, these units pass their results to the subsequent units forming a **pipeline structure** (figure 2.2). When we analyzed the phases needed for image synthesis (geometric manipulations, scan conversion and pixel operations etc.), we concluded that the algorithms, the ba-
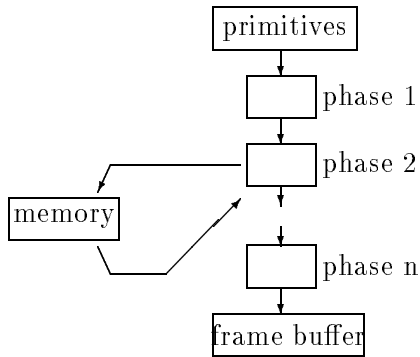
Figure 2.2: Pipeline architecture

sic data types and the speed requirements are very different in these
phases, thus this pipeline architecture makes it possible to use hard-
ware units optimized for the operations of the actual phase. The
pipeline is really effective if the data are moving in a single direction
in it. Thus, when a primitive is processed by a given phase, subse-
quent primitives can be dealt with by the previous phases and the
previous primitives by the subsequent phases. This means that an n
phase pipeline can deal with n number of primitives at the same time.
If the different phases require approximately the same amount of time
to process a single primitive, then the processing speed is increased
by n times in the pipeline architecture. If the different phases need
a different amount of time, then the slowest will determine the over-
all speed. Thus balancing the different phases is a crucial problem.
This problem cannot be solved in an optimal way for all the different
primitives because the "computational requirements" of a primitive
in the different phases depend on different factors. Concerning geo-
metric manipulations, the complexity of the calculation is determined
by the number of vertices in a polygon mesh representation, while the
complexity of pixel manipulations depends on the number of pixels
covered by the projected polygon mesh. Thus, the pipeline can only
be balanced for polygons of a given projected size. This optimal size
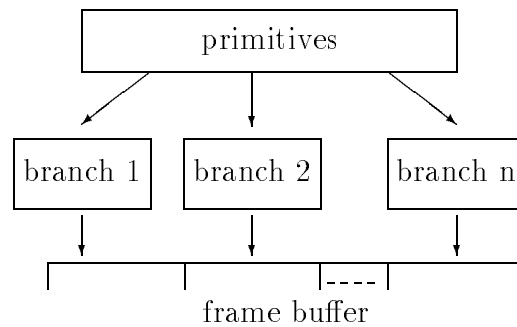must be determined by analyzing the "real applications".

Figure 2.3: Image parallel architecture

2. **Image space** or **pixel oriented parallelization** allocates different
   hardware units for those calculations which generate the color of a
   given subset of pixels (figure 2.3). Since any primitive may affect
   any pixel, the parallel branches of computation must get all primi-
   tives. The different branches realize the very same algorithm including
   all steps of image generation. Algorithms which have computational
   complexity proportional to the number of pixels can benefit from this
   architecture, because each branch works on fewer pixels than the num-
   ber of pixels in the frame buffer. Those algorithms, however, whose
   complexities are independent of the number of pixels (but usually pro-
   portional to some polynomial of the number of primitives), cannot be
   speeded up in this way, since the same algorithm should be carried
   out in each branch for all the different primitives, which require the
   same time as the calculation of all primitives by a single branch. Con-
   cerning only algorithms whose complexities depend on the number of
   pixels, the balancing of the different branches is also very important.
   Balancing means that from the same set of primitives the different
   phases generate the same number of pixels and the difficulty of calcu-
   lating pixels is also evenly distributed between the branches. This can
   be achieved if the pixel space is partitioned in a way which orders ad-
   jacent pixels into different partitions, and the color of the pixels in the
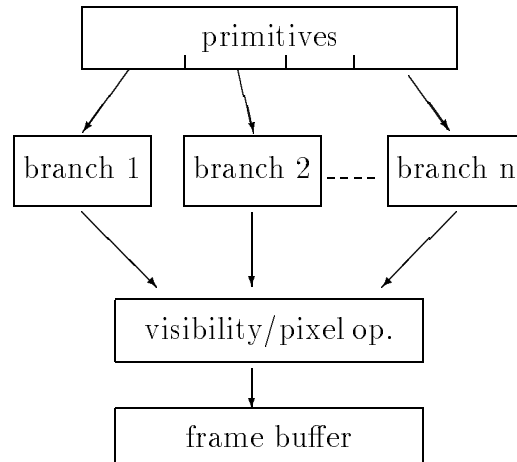   different partitions is generated by different branches of the parallel
   hardware.

Figure 2.4: Object parallel architecture

3. **Object space** or **primitive oriented parallelization** allocates different hardware units for the calculation of different subsets of primitives (figure 2.4). The different branches now get only a portion of the original primitives and process them independently. However, the different branches must meet sometimes because of the following reasons: a) the image synthesis of the different primitives cannot be totally independent because their relative position is needed for visibility calculations, and the color of a primitive may affect the color of other primitives during shading; b) any primitive can affect the color of a pixel, thus, any parallel branch may try to determine the color of the same pixel, which generates a problem that must be resolved by visibility considerations. Consequently, the parallel branches must be bundled together into a single processing path for visibility, shading and frame buffer access operations. This common point can easily be a bottleneck. This is why this approach is not as widely accepted and used as the other two.

The three alternatives discussed above represent theoretically different approaches to build a parallel system for image synthesis. In practical applications, however, combination of the different approaches can be expected to provide the best solutions. This combination can be done in different ways, which lead to different heterogeneous architectures. The image parallel architecture, for instance, was said to be inefficient for those methods which are independent of the number of pixels. The first steps of image synthesis, including geometric manipulations, are typically such methods, thus it is worth doing them before the parallel branching of the computation usually by an initial pipeline. Inside the parallel branches, on the other hand, a sequence of different operations must be executed, which can be well done in a pipeline. The resulting architecture starts with a single pipeline which breaks down into several pipelines at some stage.

The analysis of the speed requirements in the different stages of a pipeline can lead to a different marriage between pipeline and image parallel architectures. Due to the fact that a primitive usually covers many pixels when projected, the time allowed for a single data element decreases drastically between geometric manipulations, scan conversion, pixel operations and frame buffer access. As far as scan conversion and pixel operations are concerned, their algorithms are usually simple and can be realized by a special digital hardware that can cope with the high speed requirements. The speed of the frame buffer access step, however, is limited by the access time of commercial memories, which is much less than needed by the performance of other stages. Thus, frame buffer access must be speeded up by parallelization, which leads to an architecture that is basically a pipeline but at some final stage it becomes an image parallel system.

2.3 Hardware realization of graphics algorithms

In this section the general aspects of the hardware realization of graphics, mostly scan conversion algorithms are discussed. Strictly speaking, hardware realization means a special, usually synchronous, digital network designed to determine the pixel data at the speed of its clock signal.

In order to describe the difficulty of the realization of a function as a combinational network by a given component set, the measure, called **combinational complexity** or combinational realizability complexity, is introduced:

Let f be a finite valued function on the domain of a subset of natural numbers $0, 1 \ldots N$. By definition, the combinational complexity of f is D if the minimal combinational realization of f, containing no feedback, consists of D devices from the respective component set.

One possible respective component set contains NAND gates only, another covers the functional elements of MSI and LSI circuits, including:

1. **Adders, combinational arithmetic/logic units (say 32 bits)** which can execute arithmetical operations.

2. **Multiplexers** which are usually responsible for the *then ... else ...* branching of conditional operations.

3. **Comparators** which generate logic values for *if* type decisions.

4. **Logic gates** which are used for simple logic operations and decisions.

5. **Encoders, decoders and memories of reasonable size (say 16 address bits)** which can realize arbitrary functions having small domains.

The requirement that the function should be integer valued and should have integer domain would appear to cause serious limitation from the point of view of computer graphics, but in fact it does not, since negative, fractional and floating point numbers are also represented in computers by binary combinations which can be interpreted as a positive integer code word in a binary number system.

2.3.1   Single-variate functions

Suppose that functions $f_1(k), f_2(k), \ldots, f_n(k)$ having integer domain have to be computed for the integers in an interval between k_s and k_e.

A computer program carrying out this task might look like this:

```
for k = k_s to k_e do
    F_1 = f_1(k); F_2 = f_2(k); ... F_n = f_n(k);
    write( k, F_1, F_2, ... F_n );
endfor
```

If all the $f_i(k)$-s had low combinational complexity, then an independent combinational network would be devoted to each of them, and a separate hardware counter would generate the consecutive k values, making the hardware realization complete. This works for many pixel level operations, but usually fails for scan conversion algorithms due to their high combinational complexity.

Fortunately, there is a technique, called the **incremental concept**, which has proven successful for many scan conversion algorithms. According to the incremental concept, in many cases it is much simpler to calculate the value $f(k)$ from $f(k-1)$ instead of using only k, by a function of increment, $\mathcal{F}$:

$$f(k) = \mathcal{F}(f(k-1), k). \tag{2.20}$$

If this $\mathcal{F}$ has low complexity, it can be realized by a reasonably simple combinational network. In order to store the previous value $f(k-1)$, a register has to be allocated, and a counter has to be used to generate the consecutive values of k and stop the network after the last value has been computed. This consideration leads to an architecture of figure 2.5.

What happens if even $\mathcal{F}$ has too high complexity inhibiting its realization by an appropriate combinational circuit? The incremental concept might be applied to $\mathcal{F}$ as well, increasing the number of necessary temporary registers, but hopefully simplifying the combinatorial part, and that examination can also be repeated recursively if the result is not satisfactory. Finally, if this approach fails, we can turn to the simplification of the algorithm, or can select a different algorithm altogether.

Generally, the derivation of $\mathcal{F}$ requires heuristics, the careful examination and possibly the transformation of the mathematical definition or the computer program of $f(k)$. Systematic approaches, however, are available if $f(k)$ can be regarded as the restriction of a differentiable real function $f_r(r)$ to integers both in the domain and in the value set, since in this case
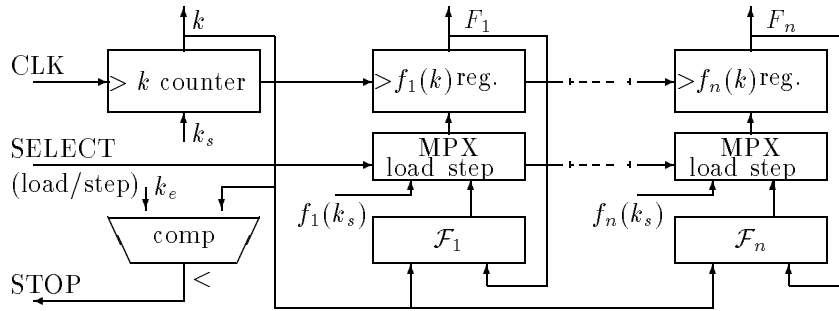
Figure 2.5: General architecture implementing the incremental concept

$f_r(k)$ can be approximated by **Taylor's series** around $f_r(k-1)$:

$$f_r(k) \approx f_r(k-1) + \frac{df_r}{dk}\Big|_{k-1} \cdot \Delta k = f_r(k-1) + f_r'(k-1) \cdot 1 \qquad (2.21)$$

The only disappointing thing about this formula is that $f_r'(k-1)$ is usually not an integer, nor is $f_r(k-1)$, and it is not possible to ignore the fractional part, since the incremental formula will accumulate the error to an unacceptable degree. The values of $f_r(k)$ should rather be stored temporarily in a register as a real value, the computations should be carried out on real numbers, and the final $f(k)$ should be derived by finding the nearest integer from $f_r(k)$. The realization of floating point arithmetic is not at all simple; indeed its high combinational complexity makes it necessary to get rid of the floating point numbers. Non-integers, fortunately, can also be represented in **fixed point form** where the low b_F bits of the code word represent the fractional part, and the high b_I bits store the integer part. From a different point of view, a code word having binary code C represents the real number $C \cdot 2^{-b_F}$. Since fixed point fractional numbers can be handled in the same way as integers in addition, subtraction, comparison and selection (not in division or multiplication where they have to be shifted after the operation), and truncation is simple in the above component set, they do not need any extra calculation.

Let us devote some time to the determination of the length of the register needed to store $f_r(k)$. Concerning the integer part, $f(k)$, the truncation of

$f_r(k)$ may generate integers from 0 to N, requiring $b_I > \log_2 N$. The number
of bits in the fractional part has to be set to avoid incorrect $f(k)$ calculations
due to the cumulative error in $f_r(k)$. Since the maximum length of the
iteration is N if $k_s = 0$ and $k_e = N$, and the maximum error introduced
by a single step of the iteration is less than 2^{-b_F}, the cumulative error is
maximum $N \cdot 2^{-b_F}$. Incorrect calculation of $f(k)$ is avoided if the cumulative
error is less than 1:

$$N \cdot 2^{-b_F} < 1 \implies b_F > \log_2 N. \tag{2.22}$$

Since the results are expected in integer form they must be converted to
integers at the final stage of the calculation. The Round function finding
the nearest integer for a real number, however, has high combinational
complexity. Fortunately, the Round function can be replaced by the Trunc
function generating the integer part of a real number if 0.5 is added to the
number to be converted. The implementation of the Trunc function poses
no problem for fixed point representation, since just the bits corresponding
to the fractional part must be neglected. This trick can generally be used
if we want to get rid of the Round function.

The proposed approach is especially efficient if the functions to be cal-
culated are linear, since that makes $f'(k - 1) = \delta f$ a constant parameter,
resulting in the network of figure 2.6. Note that the hardware consists of
similar blocks, called interpolators, which are responsible for the generation
of a single output variable.

The transformed program for linear functions is:

$F_1 = f_1(k_s) + 0.5; F_2 = f_2(k_s) + 0.5; \ldots F_n = f_n(k_s) + 0.5;$
$\delta f_1 = f_1'(k); \delta f_2 = f_2'(k); \ldots \delta f_n = f_n'(k);$
for $k = k_s$ **to** k_e **do**
    write($k, \mathrm{Trunc}(F_1), \mathrm{Trunc}(F_2), \ldots, \mathrm{Trunc}(F_n)$);
    $F_1 \mathrel{+}= \delta f_1; F_2 \mathrel{+}= \delta f_2; \ldots F_n \mathrel{+}= \delta f_n;$
endfor

The simplest example of the application of this method is the **DDA line
generator** (DDA stands for Digital Differential Analyzer which means ap-
proximately the same as the incremental method in this context). For
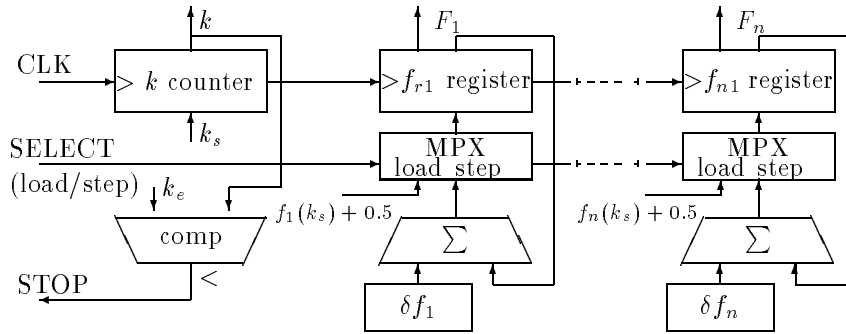notational simplicity, suppose that the generator has to work for those

Figure 2.6: Hardware for linear functions

(x_1, y_1, x_2, y_2) line segments which satisfy:

$$x_1 \leq x_2, \qquad y_1 \leq y_2, \qquad x_2 - x_1 \geq y_2 - y_1. \tag{2.23}$$

Line segments of this type can be approximated by $n = x_2 - x_1 + 1$ pixels having consecutive x coordinates. The y coordinate of the pixels can be calculated from the equation of the line:

$$y = \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1) + x_1 = m \cdot x + b. \tag{2.24}$$

Based on this formula, the algorithm needed to draw a line segment is:

> **for** $x = x_1$ **to** x_2 **do**
> $y = \mathrm{Round}(m \cdot x + b)$;
>   **write**(x, y, color);
> **endfor**

The function $f(x) = \mathrm{Round}(m \cdot x + b)$ contains multiplication, non-integer addition, and the Round operation to find the nearest integer, resulting in a high value of combinational complexity. Fortunately the incremental concept can be applied since it can be regarded as the truncation of the real-valued, differentiable function:

$$f_r(x) = m \cdot x + b + 0.5 \tag{2.25}$$

Since f_r is differentiable, the incremental formula is:

$$f_r(x) = f_r(x) + f'_r(x-1) = f_r(x) + m. \qquad (2.26)$$

The register storing $f_r(x)$ in fixed point format has to have more than $\log_2 N$ integer and more than $\log_2 N$ fractional bits, where N is the length of the longest line segment. For a display of 1280×1024 pixel resolution a 22 bit long register is required if it can be guaranteed by a previous clipping algorithm that no line segments will have coordinates outside the visible region of the display. From this point of view, clipping is not only important in that it speeds up the image synthesis by removing invisible parts, but it is also essential because it ensures the avoidance of overflows in scan conversion hardware working with fixed point numbers.
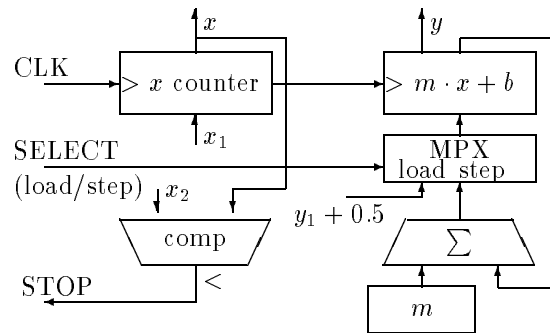
Figure 2.7: DDA line generator

The slope of the line $m = (y_2 - y_1)/(x_2 - x_1)$ has to be calculated only once and before inputting it into the hardware.

This example has confirmed that the hardware implementation of linear functions is a straightforward process, since it could remove all the multiplications and divisions from the inner cycle of the algorithm, and it requires them in the initialization phase only. For those linear functions where the fractional part is not relevant for the next phases of the image generation and $|\delta f| \leq 1$, the method can be even further optimized by reducing the computational burden of the initialization phase as well.

If the fractional part is not used later on, its only purpose is to determine when the integer part has to be incremented (or decremented) due to over-flow caused by the cumulative increments δf. Since $\delta f \leq 1$, the maximum increase or decrease in the integer part must necessarily also be 1. From this perspective, the fractional part can also be regarded as an error value showing how accurate the integer approximation is. The error value, however, is not necessarily stored as a fractional number. Other representations, not requiring divisions during the initialization, can be found, as suggested by the method of **decision variables**.

Let the fractional part of f_r be fract and assume that the increment δf is generated as a rational number defined by a division whose elimination is the goal of this approach:

$$\delta f = \frac{K}{D}.\tag{2.27}$$

The overflow of fract happens when fract $+\ \delta f > 1$. Let the new error variable be $E = 2D \cdot (\text{fract} - 1)$, requiring the following incremental formula for each cycle:

$$E(k) = 2D \cdot (\text{fract}(k) - 1) = 2D \cdot ([\text{fract}(k-1) + \delta f] - 1) = E(k-1) + 2K.\tag{2.28}$$

The recognition of overflow is also easy:

$$\text{fract}(k) \geq 1.0 \quad \Longrightarrow \quad E(k) \geq 0\tag{2.29}$$

If overflow happens, then the fractional part is decreased by one, since the bit which has the first positional significance overflowed to the integer part:

$$\text{fract}(k) = [\text{fract}(k-1) + \delta f] - 1 \quad \Longrightarrow \quad E(k) = E(k-1) + 2(K - D).\tag{2.30}$$

Finally, the initial value of E comes from the fact that fract has to be initialized to 0.5, resulting in:

$$\text{fract}(0) = 0.5 \quad \Longrightarrow \quad E(0) = -D.\tag{2.31}$$

Examining the formulae of E, we can conclude that they contain integer additions and comparisons, eliminating all the non-integer operations. Clearly, it is due to the multiplication by $2D$, where D compensates for the

fractional property of $\delta f = K/D$ and 2 compensates for the 0.5 initial value responsible for replacing Round by Trunc.

The first line generator of this type has been proposed by Bresenham [Bre65].

Having made the substitutions, $K = y_2 - y_1$ and $D = x_2 - x_1$, the code of the algorithm in the first octant of the plane is:

> **BresenhamLine**(x_1, y_1, x_2, y_2)
>> $\Delta x = x_2 - x_1; \Delta y = y_2 - y_1;$
>> $E = -\Delta x;$
>> $dE^+ = 2(\Delta y - \Delta x); dE^- = 2\Delta y;$
>> $y = y_1;$
>> **for** $x = x_1$ **to** x_2 **do**
>>> **if** $E \leq 0$ **then** $E \mathrel{+}= dE^-;$
>>> **else** $\qquad\qquad E \mathrel{+}= dE^+; y{+}{+};$
>>> **write**$(x, y, \text{color});$
>> **endfor**

2.3.2   Multi-variate functions

The results of the previous section can be generalized to higher dimensions, but, for the purposes of this book, only the two-variate case has any practical importance, and this can be formulated as follows:
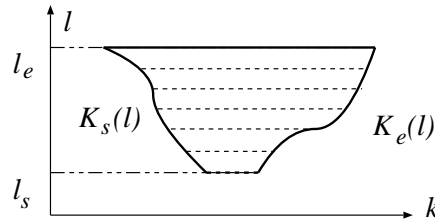
Figure 2.8: The domain of the two-variate functions

Let a set of two-variate functions be $f_1(k, l), f_2(k, l), \ldots, f_n(k, l)$ and suppose we have to compute them for domain points (figure 2.8):

$$S = \{(k, l) \mid l_s \leq l \leq l_e, \; K_s(l) \leq k \leq K_e(l)\}. \qquad (2.32)$$

A possible program for this computation is:

```
for l = l_s to l_e do
    for k = K_s(l) to K_e(l) do
        F_1 = f_1(k, l); F_2 = f_2(k, l); ... F_n = f_n(k, l);
        write( k, l, F_1, F_2, ... F_n );
    endfor
endfor
```

Functions f, K_s, K_e are assumed to be the truncations of real valued, differentiable functions to integers. Incremental formulae can be derived for these functions relying on Taylor's approximation:

$$f_r(k+1, l) \approx f_r(k, l) + \frac{\partial f_r(k, l)}{\partial k} \cdot 1 = f_r(k, l) + \delta f^k(k, l), \qquad (2.33)$$

$$K_s(l+1) \approx K_s(l) + \frac{dK_s(l)}{dl} \cdot 1 = K_s(l) + \delta K_s(l), \qquad (2.34)$$

$$K_e(l+1) \approx K_e(l) + \frac{dK_e(l)}{dl} \cdot 1 = K_e(l) + \delta K_e(l). \qquad (2.35)$$

The increments of $f_r(k, l)$ along the boundary curve $K_s(l)$ is:

$$f_r(K_s(l+1), l+1) \approx f_r(K_s(l), l) + \frac{df_r(K_s(l), l)}{dl} = f_r(K_s(l), l) + \delta f^{l,s}(l).$$
$$(2.36)$$

These equations are used to transform the original program computing f_i-s:

```
S = K_s(l_s) + 0.5; E = K_e(l_s) + 0.5;
F_1^s = f_1(K_s(l_s), l_s) + 0.5; ... F_n^s = f_n(K_s(l_s), l_s) + 0.5;
for l = l_s to l_e do
    F_1 = F_1^s; F_2 = F_2^s; ... F_n = F_n^s;
    for k = Trunc(S) to Trunc(E) do
        write( k, l, Trunc(F_1), Trunc(F_2), ..., Trunc(F_n) );
        F_1 += δf_1^k; F_2 += δf_2^k; ... F_n += δf_n^k;
    endfor
    F_1^s += δf_1^{l,s}; F_2^s += δf_2^{l,s}; ... F_n^s += δf_n^{l,s};
    S += δK_s; E += δK_e;
endfor
```
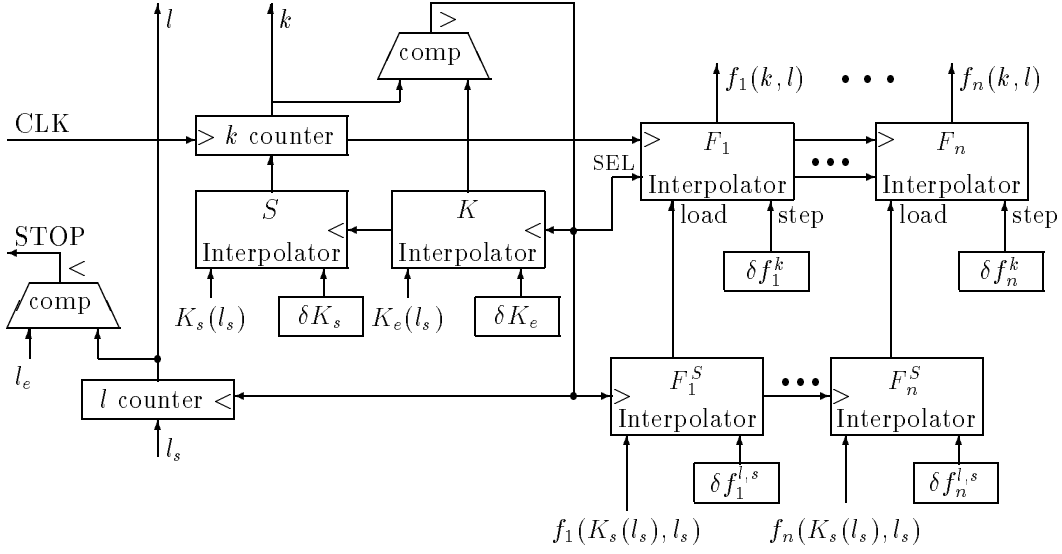
Figure 2.9: Hardware realization of two-variate functions

Concerning the hardware realization of this transformed program, a two level hierarchy of interpolators should be built. On the lower level interpolators have to be allocated for each F_i, which are initialized by a respective higher level interpolator generating F_i^s. The counters controlling the operation also form a two-level hierarchy. The higher level counter increments two additional interpolators, one for start position S, and one for end condition E, which, in turn, serve as start and stop control values for the lower level counter. Note that in the modified algorithm the longest path where the round-off errors can accumulate consists of

$$\max_l\{l - l_s + K_e(l) - K_s(l)\} \leq P_k + P_l$$

steps, where P_k and P_l are the size of the domain of k and l respectively. The minimum length of the fractional part can be calculated by:

$$b_F > \log_2(P_k + P_l). \tag{2.37}$$

A hardware implementation of the algorithm is shown in figure 2.9.

2.3.3   Alternating functions

Alternating functions have only two values in their value set, which alternates according to a selector function. They form an important set of non-differentiable functions in computer graphics, since pattern generators responsible for drawing line and tile patterns and characters fall into this category. Formally an alternating function is:

$$f(k) = F(s(k)), \qquad s(k) \in \{0, 1\}. \tag{2.38}$$

Function F may depend on other input parameters too, and it usually has small combinational complexity. The selector $s(k)$ may be periodic and is usually defined by a table. The hardware realization should, consequently, find the kth bit of the definition table to evaluate $f(k)$. A straightforward way to do that is to load the table into a shift register (or into a circular shift register if the selector is periodic) during initialization, and in each iteration select the first bit to provide $s(k)$ and shift the register to prepare for the next k value.
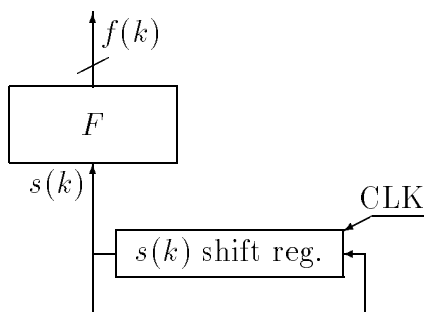
Figure 2.10: Hardware for alternating functions

Alternating functions can also be two-dimensional, for example, to generate tiles and characters. A possible architecture would require a horizontal and a vertical counter, and a shift register for each row of the pattern. The vertical counter selects the actual shift register, and the horizontal counter, incremented simultaneously with the register shift, determines when the vertical counter has to be incremented.

Chapter 3

PHYSICAL MODEL OF 3D IMAGE SYNTHESIS

3.1 Definition of color

Light is an electromagnetic wave, and its color is determined by the eye's perception of its spectral energy distribution. In other words, the color is determined by the frequency spectrum of the incoming light. Due to its internal structure, the eye is a very poor spectrometer since it actually samples and integrates the energy in three overlapping frequency ranges by three types of photopigments according to a widely accepted (but also argued) model of the eye. As a consequence of this, any color perception can be represented by a point in a three-dimensional space, making it possible to define color perception by three scalars (called **tristimulus** values) instead of complete functions.

A convenient way to define the axes of a coordinate system in the space of color sensations is to select three wavelengths where one type of photopigment is significantly more sensitive than the other two. This is the method devised by Grassmann, who also specified a criterion for separating the three representative wavelengths. He states in his famous laws that the representative wavelengths should be selected such that no one of them can be matched by the mixture of the other two in terms of color sensation. (This criterion is similar to the concept of linear independence.)

An appropriate collection of representative wavelengths is:

$$\lambda_{\text{red}} = 700 \text{ nm}, \quad \lambda_{\text{green}} = 561 \text{ nm}, \quad \lambda_{\text{blue}} = 436 \text{ nm}. \tag{3.1}$$

Now let us suppose that monochromatic light of wavelength λ is perceived by the eye. The equivalent portions of red, green and blue light, or (r, g, b) tristimulus values, can be generated by three **color matching functions** $(r(\lambda), g(\lambda)$ and $b(\lambda))$ which are based on physiological measurements.
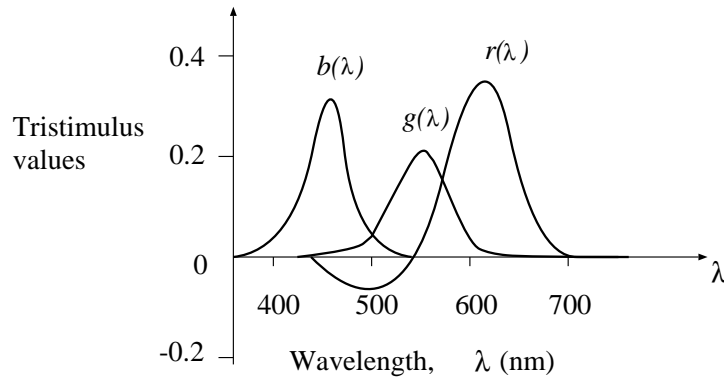
Figure 3.1: Color matching functions $r(\lambda)$, $g(\lambda)$ and $b(\lambda)$

If the perceived color is not monochromatic, but is described by an $L(\lambda)$ distribution, the tristimulus coordinates are computed using the assumption that the sensation is produced by an additive mixture of elemental monochromatic components:

$$r = \int_\lambda L(\lambda) \cdot r(\lambda) \, d\lambda, \qquad g = \int_\lambda L(\lambda) \cdot g(\lambda) \, d\lambda, \qquad b = \int_\lambda L(\lambda) \cdot b(\lambda) \, d\lambda. \tag{3.2}$$

Note the negative section of $r(\lambda)$ in figure 3.1. It means that not all the colors can be represented by positive (r, g, b) values, hence there are colors which cannot be produced, only approximated, on the computer screen.

This negative matching function can be avoided by careful selection of the axes in the color coordinate system, and in fact, in 1931 another standard, called the **CIE XYZ** system, was defined which has only positive weights [WS82].

For computer generated images, the color sensation of an observer watching a virtual world on the screen must be approximately equivalent to the color sensation obtained in the real world. If two energy distributions are associated with the same tristimulus coordinates, they produce the same color sensation, and are called **metamers**.

In computer monitors and on television screens red, green and blue phosphors can be stimulated to produce red, green and blue light. The objective, then, is to find the necessary stimulus to produce a metamer of the real energy distribution of the light. This stimulus can be controlled by the (R, G, B) values of the actual pixel. These values are usually positive numbers in the range of $[0...255]$ if 8 bits are available to represent them.

Let the distribution of the energy emitted by red, green and blue phosphors be $P_R(\lambda, R)$, $P_G(\lambda, G)$ and $P_B(\lambda, B)$, respectively, for a given (R, G, B) pixel color. Since the energy distribution of a type of phosphor is concentrated around wavelength λ_{red}, λ_{green} or λ_{blue}, the tristimulus coordinates of the produced light will look like this:

$$r = \int_\lambda (P_R + P_G + P_B) \cdot r(\lambda) \, d\lambda \approx \int_\lambda P_R(\lambda, R) \cdot r(\lambda) \, d\lambda = p_R(R), \quad (3.3)$$

$$g = \int_\lambda (P_R + P_G + P_B) \cdot g(\lambda) \, d\lambda \approx \int_\lambda P_G(\lambda, G) \cdot g(\lambda) \, d\lambda = p_G(G), \quad (3.4)$$

$$b = \int_\lambda (P_R + P_G + P_B) \cdot b(\lambda) \, d\lambda \approx \int_\lambda P_B(\lambda, B) \cdot b(\lambda) \, d\lambda = p_B(B). \quad (3.5)$$

Expressing the necessary R, G, B values, we get:

$$R = p_R^{-1}(r), \qquad G = p_G^{-1}(g), \qquad B = p_B^{-1}(b). \qquad (3.6)$$

Unfortunately p_R, p_G and p_B are not exactly linear functions of the calculated R, G and B values, due to the non-linearity known as γ-distortion of color monitors, but follow a $\text{const} \cdot N^\gamma$ function, where N is the respective R, G or B value. In most cases this non-linearity can be ignored, allowing $R = r$, $G = g$ and $B = b$. Special applications, however, require compensation for this effect, which can be achieved by rescaling the R, G, B values by appropriate lookup tables according to functions p_R^{-1}, p_G^{-1} and p_B^{-1}. This method is called γ-**correction**.

Now we can focus on the calculation of the (r, g, b) values of the color perceived by the eye or camera through an (X,Y) point in the window.

According to the laws of optics, the virtual world can be regarded as a system that transforms a part of the light energy of the lightsources $(P_{in}(\lambda))$ into a light beam having energy distribution $P_{XY}(\lambda)$ and going to the camera through pixel (X,Y). Let us denote the transformation by functional L:

$$P_{XY}(\lambda) = L(P_{in}(\lambda)). \tag{3.7}$$

A tristimulus color coordinate, say r, can be determined by applying the appropriate matching function:

$$r_{XY} = \int_{\lambda} P_{XY}(\lambda) \cdot r(\lambda) \, d\lambda = \int_{\lambda} L(P_{in}(\lambda)) \cdot r(\lambda) \, d\lambda. \tag{3.8}$$

In order to evaluate this formula numerically, $L(P_{in}(\lambda))$ is calculated in discrete points $\lambda_1, \lambda_2, ..., \lambda_n$, and rectangular or trapezoidal integration rule is used:

$$r_{XY} \approx \sum_{i=1}^{n} L(P_{in}(\lambda_i)) \cdot r(\lambda_i) \cdot \Delta\lambda_i. \tag{3.9}$$

Similar equations can be derived for the other two tristimulus values, g and b. These equations mean that the calculation of the pixel colors requires the solution of the shading problem, or evaluating the L functional, for n different wavelengths independently, then the r, g and b values can be determined by summation of the results weighted by their respective matching functions. Examining the shape of matching functions, however, we can conclude that for many applications an even more drastic approximation is reasonable, where the matching function is replaced by a function of rectangular shape:

$$r(\lambda) \approx \hat{r}(\lambda) = \begin{cases} r_{max} \text{ if } \lambda_{red} - \Delta\lambda_{red}/2 \leq \lambda \leq \lambda_{red} + \Delta\lambda_{red}/2 \\ \\ 0 \text{ otherwise} \end{cases} \tag{3.10}$$

Using this approximation, and assuming L to be linear in terms of the energy (as it really is) and $L(0) = 0$, we get:

$$r_{XY} \approx \int_{\lambda} L(P_{in}(\lambda)) \cdot \hat{r}(\lambda) \, d\lambda = L(\int_{\lambda} P_{in}(\lambda) \cdot \hat{r}(\lambda) \, d\lambda) = L(r_{in}), \tag{3.11}$$

where r_in is the first tristimulus coordinate of the energy distribution of the lightsources $(P_\mathrm{in}(\lambda))$.

This means that the tristimulus values of the pixel can be determined from the tristimulus values of the lightsources. Since there are three tristimulus coordinates (blue and green can be handled exactly the same way as red) the complete shading requires independent calculations for only three wavelengths. If more accurate color reproduction is needed, equation 3.9 should be applied to calculate r, g and b coordinates.

3.2   Light and surface interaction

Having separated the color into several (mainly three) representative frequencies, the problem to be solved is the calculation of the energy reaching the camera from a given direction, i.e. through a given pixel, taking into account the optical properties of the surfaces and the lightsources in the virtual world. Hereinafter, monochromatic light of a representative wavelength λ will be assumed, since the complete color calculation can be broken down to these representative wavelengths. The parameters of the equations usually depend on the wavelength, but for notational simplicity, we do not always include the λ variable in them.
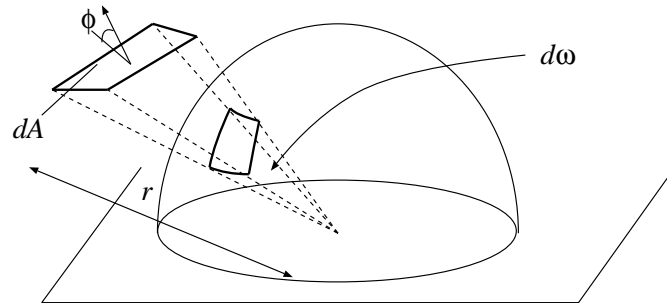
Figure 3.2: Definition of the solid angle

The directional property of the energy emission is described in a so-called **illumination hemisphere** which contains those solid angles to where the surface point can emit energy. By definition, a **solid angle** is a cone or a pyramid, with its size determined by its subtended area of a unit sphere

centered around the apex (figure 3.2). The solid angle, in which a differential dA surface can be seen from point $\vec{p}$, is obviously the projected area per the square of the distance of the surface. If the angle between the surface normal of dA and the directional vector from dA to $\vec{p}$ is ϕ, and the distance from dA to $\vec{p}$ is r, then this solid angle is:

$$d\omega = \frac{dA \cdot \cos \phi}{r^2}. \tag{3.12}$$

The intensity of the energy transfer is characterized by several metrics in computer graphics depending on whether or not the directional and positional properties are taken into account.

The light power or **flux** Φ is the energy radiated through a boundary per unit time over a given range of the spectrum (say $[\lambda, \lambda + d\lambda]$).

The **radiant intensity**, or **intensity** I for short, is the differential light flux leaving a surface element dA in a differential solid angle $d\omega$ per the projected area of the surface element and the size of the solid angle. If the angle of the surface normal and the direction of interest is ϕ, then the projected area is $dA \cdot \cos \phi$, hence the intensity is:

$$I = \frac{d\Phi(d\omega)}{dA \cdot d\omega \cdot \cos \phi}. \tag{3.13}$$

The total light flux radiated through the hemisphere centered over the surface element dA per the area of the surface element is called the **radiosity** B of surface element dA.
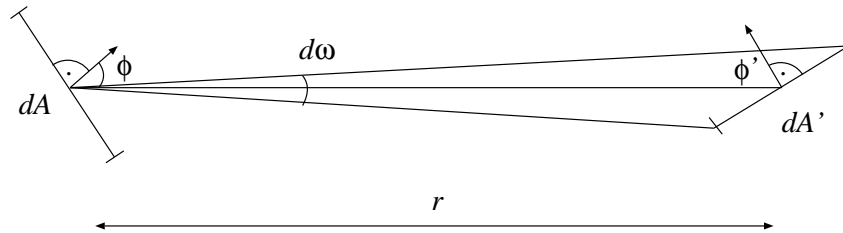

Figure 3.3: Energy transfer between two differential surface elements

Having introduced the most important metrics, we turn to their determination in the simplest case, where there are only two differential surface

elements in the 3D space, one (dA) emits light energy and the other (dA') absorbs it (figure 3.3). If dA' is visible from dA in solid angle $d\omega$ and the radiant intensity of the surface element dA is $I(d\omega)$ in this direction, then the flux leaving dA and reaching dA' is:

$$d\Phi = I(d\omega) \cdot dA \cdot d\omega \cdot \cos\phi. \qquad (3.14)$$

according to the definition of the radiant intensity. Expressing the solid angle by the projected area of dA', we get:

$$d\Phi = I \cdot \frac{dA \cdot \cos\phi \cdot dA' \cdot \cos\phi'}{r^2}. \qquad (3.15)$$

This formula is called the **fundamental law of photometry**.
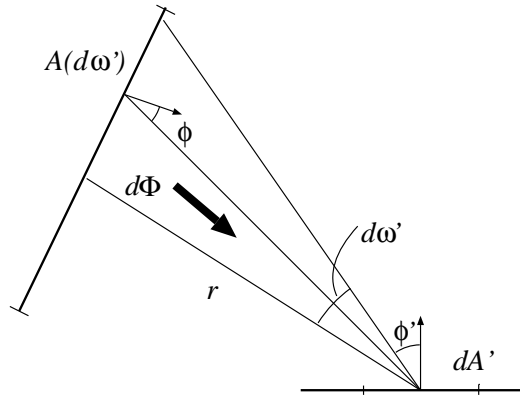
Figure 3.4: Radiation of non-differential surfaces

Real situations containing not differential, but finite surfaces can be discussed using as a basis this very simple case (figure 3.4). Suppose there is a finite radiating surface (A), and we are interested in its energy reaching a dA' element of another surface in the solid angle $d\omega'$. The area of the radiating surface visible in the solid angle $d\omega'$ is $A(d\omega') = r^2 \cdot d\omega'/\cos\phi$, so the flux radiating dA' from the given direction will be independent of the position and orientation of the radiating surface and will depend on its intensity only, since:

$$d\Phi = I \cdot \frac{A(d\omega') \cdot \cos\phi \cdot dA' \cdot \cos\phi'}{r^2} = I \cdot dA' \cdot \cos\phi' \cdot d\omega' = \text{const} \cdot I. \quad (3.16)$$

Similarly, if the flux going through a pixel to the camera has to be calculated (figure 3.5), the respective solid angle is:

$$d\omega_{\mathrm{pix}} = \frac{dA_{\mathrm{pix}} \cdot \cos \phi_{\mathrm{pix}}}{r_{\mathrm{pix}}^2}. \tag{3.17}$$

The area of the surface fragment visible through this pixel is:

$$A(d\omega_{\mathrm{pix}}) = \frac{r^2 \cdot d\omega_{\mathrm{pix}}}{\cos \phi}. \tag{3.18}$$

Thus, the energy defining the color of the pixel is:

$$d\Phi_{\mathrm{pix}} = I \cdot \frac{A(d\omega_{\mathrm{pix}}) \cdot \cos \phi \cdot dA_{\mathrm{pix}} \cdot \cos \phi_{\mathrm{pix}}}{r^2} = I \cdot dA_{\mathrm{pix}} \cdot \cos \phi_{\mathrm{pix}} \cdot d\omega_{\mathrm{pix}} = \mathrm{const} \cdot I. \tag{3.19}$$
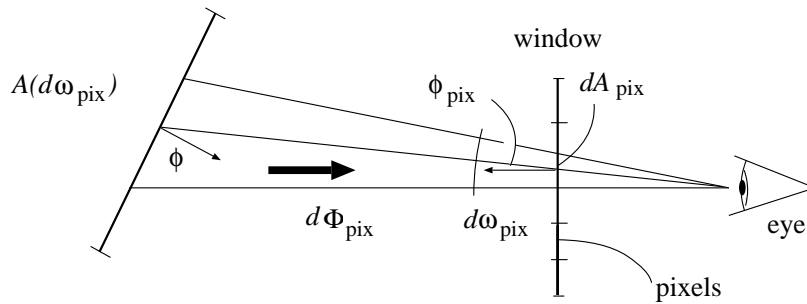
Figure 3.5: Energy transfer from a surface to the camera

Note that the intensity of a surface in a scene remains constant to an observer as he moves towards or away from the surface, since the inverse square law of the energy flux is compensated for by the square law of the solid angle subtended by the surface. Considering this property, the intensity is the best metric to work with in synthetic image generation, and we shall almost exclusively use it in this book.

In light-surface interaction the surface illuminated by an incident beam may reflect a portion of the incoming energy in various directions or it may absorb the rest. It has to be emphasized that a physically correct model must maintain energy equilibrium, that is, the reflected and the transmitted (or absorbed) energy must be equal to the incident energy.

Suppose the surface is illuminated by a beam having energy flux Φ from the differential solid angle $d\omega$. The surface splits this energy into reflected and transmitted components, which are also divided into **coherent** and **incoherent** parts.
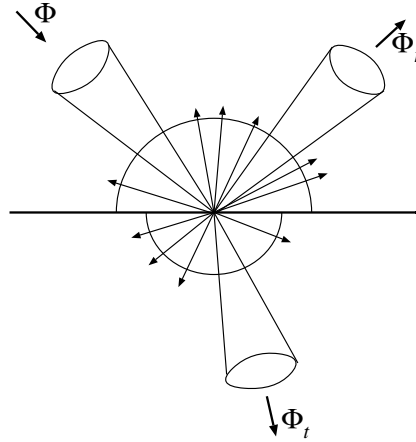

Figure 3.6: Transformation of the incident light by a surface

Optically perfect or smooth surfaces will reflect or transmit only coherent components governed by the laws of geometric optics, including the law of reflection and the Snellius–Descartes law of refraction. If the surface is optically perfect, the portions of reflection (Φ_r) and transmission (Φ_t) (figure 3.6) can be defined by the **Fresnel coefficients** F_r, F_t, namely:

$$\Phi_r = F_r \cdot \Phi, \qquad \Phi_t = F_t \cdot \Phi. \tag{3.20}$$

The energy equilibrium requires $F_t + F_r = 1$.

The incoherent components are caused by the surface irregularities reflecting or refracting the incident light in any direction. Since the exact nature of these irregularities is not known, the incoherent component is modeled by means of probability theory. Assume that a photon comes from the direction denoted by unit vector $\vec{L}$. The event that this photon will leave the surface in the reflection or in the refraction direction being in the solid angle $d\omega$ around unit vector $\vec{V}$ can be broken down into the following mutually exclusive events:

1. if $\vec{L}$ and $\vec{V}$ obey the reflection law of geometric optics, the probability of the photon leaving the surface exactly at $\vec{V}$ is denoted by k_r.

2. if $\vec{L}$ and $\vec{V}$ obey the **Snellius–Descartes law** of refraction — that is

$$\frac{\sin \phi_{\text{in}}}{\sin \phi_{\text{out}}} = \nu,$$

   where ϕ_{in} and ϕ_{out} are the incident and refraction angles respectively and ν is the refractive index of the material — then the probability of the photon leaving the surface exactly at $\vec{V}$ is denoted by k_t.

3. The probability of incoherent reflection and refraction onto the solid angle $d\omega$ at $\vec{V}$ is expressed by the **bi-directional reflection** and **refraction functions** $R(\vec{L}, \vec{V})$ and $T(\vec{L}, \vec{V})$ respectively:

$$R(\vec{L}, \vec{V}) \cdot d\omega = \Pr\{\text{photon is reflected to } d\omega \text{ around } \vec{V} \mid \text{it comes from } \vec{L}\}, \tag{3.21}$$

$$T(\vec{L}, \vec{V}) \cdot d\omega = \Pr\{\text{photon is refracted to } d\omega \text{ around } \vec{V} \mid \text{it comes from } \vec{L}\}. \tag{3.22}$$

Note that the total bi-directional probability distribution is a mixed, discrete-continuous distribution, since the probability that the light may obey the laws of geometric optics is non-zero. The energy equilibrium guarantees that the integration of the bi-directional probability density over the whole sphere is 1.

Now we are ready to consider the inverse problem of light-surface interaction. In fact, computer graphics is interested in the radiant intensity of surfaces from various directions due to the light energy reaching the surface from remaining part of the 3D space (figure 3.7).

The light flux (Φ^{out}) leaving the surface at the solid angle $d\omega$ around $\vec{V}$ consists of the following incident light components:

1. That portion of a light beam coming from incident direction corresponding to the $\vec{V}$ reflection direction, which is coherently reflected. If that beam has flux Φ_r, then the contribution to Φ^{out} is $k_r \cdot \Phi_r^{\text{in}}$.

2. That portion of a light beam coming from the incident direction corresponding to the $\vec{V}$ refraction direction, which is coherently refracted. If that beam has flux Φ_t^{in}, then the contribution to Φ^{out} is $k_t \cdot \Phi_t^{\text{in}}$.
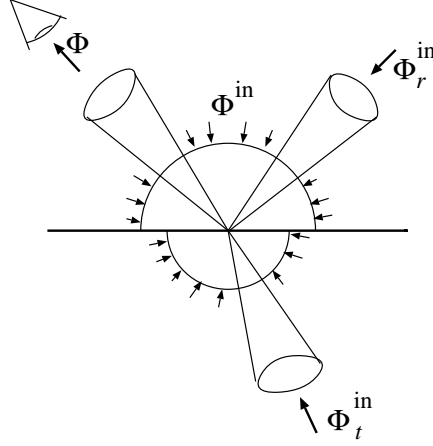
Figure 3.7: Perceived color of a surface due to incident light beams

3. The energy of light beams coming from any direction above the surface (or outside the object) and being reflected incoherently onto the given solid angle. This contribution is expressed as the integration of all the possible incoming directions $\vec{L}$ over the hemisphere above the surface:

$$\int\limits^{2\pi} (R(\vec{L}, \vec{V})\ d\omega)\Phi^{\text{in}}(\vec{L}, d\omega_{\text{in}}). \tag{3.23}$$

4. The energy of light beams coming from any direction under the surface (or from inside the object) and being refracted incoherently onto the given solid angle. This contribution is expressed as the integration of all the possible incoming directions $\vec{L}$ over the hemisphere under the surface:

$$\int\limits^{2\pi} (T(\vec{L}, \vec{V})\ d\omega)\Phi^{\text{in}}(\vec{L}, d\omega_{\text{in}}). \tag{3.24}$$

5. If the surface itself emits energy, that is, if it is a lightsource, then the emission also contributes to the output flux:

$$\Phi^e(\vec{V}). \tag{3.25}$$

Adding the possible contributions we get:

$$\Phi^{\text{out}} = \Phi^e + k_r \cdot \Phi_r^{\text{in}} + k_t \cdot \Phi_t^{\text{in}} +$$

$$\int\limits_0^{2\pi} (R(\vec{L}, \vec{V}) \, d\omega)\Phi^{\text{in}}(\vec{L}, d\omega_{\text{in}}) + \int\limits_0^{2\pi} (T(\vec{L}, \vec{V}) \, d\omega)\Phi^{\text{in}}(\vec{L}, d\omega_{\text{in}}). \qquad (3.26)$$
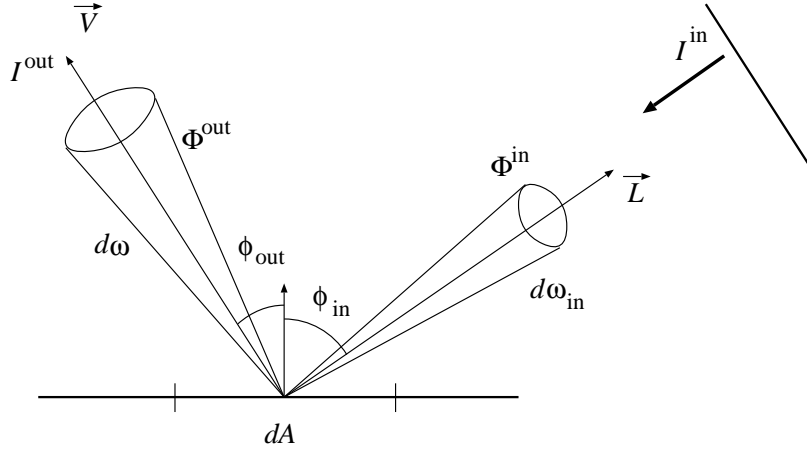

Figure 3.8: Interdependence of intensity of surfaces and the energy flux

Recall that the radiant intensity is the best metric to deal with, so this equation is converted to contain the intensities of surfaces involved. Using the notations of figure 3.8 and relying on equation 3.16, the flux of the incident light beam can be expressed by the radiant intensity of the other surface (I^{in}) and the parameters of the actual surface thus:

$$\Phi^{\text{in}}(\vec{L}, d\omega_{\text{in}}) = I^{\text{in}} \cdot dA \cdot \cos\phi_{\text{in}} \cdot d\omega_{\text{in}}. \qquad (3.27)$$

Applying this equation for Φ_r^{in} and Φ_t^{in} the intensities of surfaces in the reflection direction (I_r^{in}) and in the refraction direction (I_t^{in}) can be expressed.

The definition of the radiant intensity (equation 3.13) expresses the intensity of the actual surface:

$$\Phi^{\text{out}}(\vec{V}, d\omega) = I^{\text{out}} \cdot dA \cdot \cos\phi_{\text{out}} \cdot d\omega. \qquad (3.28)$$

Substituting these terms into equation 3.26 and dividing both sides by $dA \cdot d\omega \cdot \cos \phi_{\text{out}}$ we get:

$$I^{\text{out}} = I_e + k_r \cdot I_r \cdot \frac{\cos \phi_r \cdot d\omega_r}{\cos \phi_{\text{out}} \cdot d\omega} + k_t \cdot I_t \cdot \frac{\cos \phi_t \cdot d\omega_t}{\cos \phi_{\text{out}} \cdot d\omega} +$$

$$\int\limits_0^{2\pi} I^{\text{in}}(\vec{L}) \cdot \cos \phi_{\text{in}} \cdot \frac{R(\vec{L}, \vec{V})}{\cos \phi_{\text{out}}} \, d\omega_{\text{in}} + \int\limits_0^{2\pi} I^{\text{in}}(\vec{L}) \cdot \cos \phi_{\text{in}} \cdot \frac{T(\vec{L}, \vec{V})}{\cos \phi_{\text{out}}} \, d\omega_{\text{in}}. \quad (3.29)$$

According to the reflection law, $\phi_{\text{out}} = \phi_r$ and $d\omega = d\omega_r$. If the refraction coefficient ν is about 1, then $\cos \phi_{\text{out}} \cdot d\omega \approx \cos \phi_t \cdot d\omega_t$ holds.

Using these equations and introducing $R^*(\vec{L}, \vec{V}) = R(\vec{L}, \vec{V})/\cos \phi_{\text{out}}$ and $T^*(\vec{L}, \vec{V}) = T(\vec{L}, \vec{V})/\cos \phi_{\text{out}}$, we can generate the following fundamental formula, called the **shading, rendering** or **illumination equation**:

$$I^{\text{out}} = I_e + k_r I_r + k_t I_t +$$

$$\int\limits_0^{2\pi} I^{\text{in}}(\vec{L}) \cdot \cos \phi_{\text{in}} \cdot R^*(\vec{L}, \vec{V}) \, d\omega_{\text{in}} + \int\limits_0^{2\pi} I^{\text{in}}(\vec{L}) \cdot \cos \phi_{\text{in}} \cdot T^*(\vec{L}, \vec{V}) \, d\omega_{\text{in}}. \quad (3.30)$$

Formulae of this type are called **Hall equations**. In fact, every color calculation problem consists of several Hall equations, one for each representative frequency. Surface parameters $(I_e, k_r, k_t, R^*(\vec{L}, \vec{V}), T^*(\vec{L}, \vec{V}))$ obviously vary in the different equations.

3.3   Lambert's model of incoherent reflection

The incoherent components are modeled by bi-directional densities in the Hall equation, but they are difficult to derive for real materials. Thus, we describe these bi-directional densities by some simple functions containing a few free parameters instead. These free parameters can be used to tune the surface properties to provide an appearance similar to that of real objects.

First of all, consider diffuse — optically very rough — surfaces reflecting a portion of the incoming light with radiant intensity uniformly distributed in all directions. The constant radiant intensity (I_d) of the diffuse surface lit by a collimated beam from the angle ϕ_{in} can be calculated thus:

$$I_d = \int\limits_0^{2\pi} I^{\text{in}}(\vec{L}) \cdot \cos \phi_{\text{in}} \cdot R^*(\vec{L}, \vec{V}) \, d\omega_{\text{in}}. \quad (3.31)$$
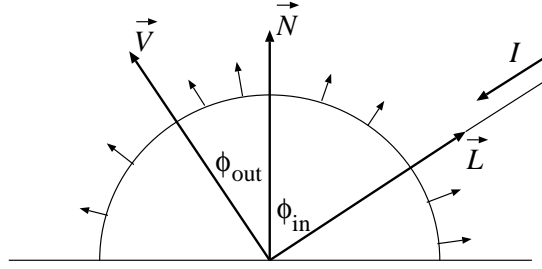
Figure 3.9: Diffuse reflection

The collimated beam is expressed as a directional delta function, $I^{\text{in}} \cdot \delta(\vec{L})$, simplifying the integral as:

$$I_d = I^{\text{in}} \cdot \cos \phi_{\text{in}} \cdot R^*(\vec{L}, \vec{V}). \qquad (3.32)$$

Since I_d does not depend on $\vec{V}$ or ϕ_{out}, the last term is constant and is called the **diffuse reflection coefficient** k_d:

$$k_d = R^*(\vec{L}, \vec{V}) = \frac{R(\vec{L}, \vec{V})}{\cos \phi_{\text{out}}}. \qquad (3.33)$$

The radiant intensity of a diffuse surface is:

$$I_d(\lambda) = I^{\text{in}}(\lambda) \cdot \cos \phi_{\text{in}} \cdot k_d(\lambda). \qquad (3.34)$$

This is **Lambert's law** of diffuse reflection. The term $\cos \phi_{\text{in}}$ can be calculated as the dot product of unit vectors $\vec{N}$ and $\vec{L}$. Should $\vec{N} \cdot \vec{L}$ be negative, the light is incident to the back of the surface, meaning it is blocked by the object. This can be formulated by the following rule:

$$I_d(\lambda) = I^{\text{in}}(\lambda) \cdot k_d(\lambda) \cdot \max\{(\vec{N} \cdot \vec{L}), 0\}. \qquad (3.35)$$

This rule makes the orientation of the surface normals essential, since they always have to point outward from the object.

It is interesting to examine the properties of the diffuse coefficient k_d. Suppose the diffuse surface reflects a fraction r of the incoming energy, while the rest is absorbed.

The following interdependence holds between k_d and r:

$$r = \frac{\Phi^{\text{out}}}{\Phi^{\text{in}}} = \frac{dA \cdot \int\limits_0^{2\pi} I_d \cdot \cos \phi_{\text{out}} \ d\omega}{dA \cdot \int\limits_0^{2\pi} I^{\text{in}} \cdot \delta(\vec{L}_{\text{in}}) \cdot \cos \phi_{\text{in}} \ d\omega_{\text{in}}} =$$

$$\frac{\int\limits_0^{2\pi} I^{\text{in}} \cdot \cos \phi_{\text{in}} \cdot k_d \cdot \cos \phi_{\text{out}} \ d\omega}{I^{\text{in}} \cdot \cos \phi_{\text{in}}} = k_d \cdot \int\limits_0^{2\pi} \cos \phi_{\text{out}} \ d\omega = k_d \cdot \pi. \qquad (3.36)$$

Note that diffuse surfaces do not distribute the light flux evenly in different directions, but follow a $\cos \phi_{\text{out}}$ function, which is eventually compensated for by its inverse in the projected area of the expression of the radiant intensity. According to equation 3.36, the k_d coefficient cannot exceed $1/\pi$ for physically correct models. In practical computations however, it is usually nearly 1, since in the applied models, as we shall see, so many phenomena are ignored that overemphasizing the computationally tractable features becomes acceptable.

Since diffuse surfaces cannot generate mirror images, they present their **"own color"** if they are lit by white light. Thus, the spectral dependence of the diffuse coefficient k_d, or the relation of k_d^{red}, k_d^{green} and k_d^{blue} in the simplified case, is primarily responsible for the surface's "own color" even in the case of surfaces which also provide non-diffuse reflections.

3.4 Phong's model of incoherent reflection

A more complex approximation of the incoherent reflection has been proposed by Phong [Pho75]. The model is important in that it also covers shiny surfaces. Shiny surfaces do not radiate the incident light by uniform intensity, but tend to distribute most of their reflected energy around the direction defined by the reflection law of geometric optics.

It would seem convenient to break down the reflected light and the bi-directional reflection into two terms; a) the diffuse term that satisfies Lambert's law and b) the specular term that is responsible for the glossy reflection concentrated around the mirror direction:

$$R(\vec{L}, \vec{V}) = R_d(\vec{L}, \vec{V}) + R_s(\vec{L}, \vec{V}), \qquad (3.37)$$

$$I^{\text{out}} = I_d + I_s = I^{\text{in}} \cdot k_d \cdot \cos \phi_{\text{in}} + I^{\text{in}} \cdot \cos \phi_{\text{in}} \cdot \frac{R_s(\vec{L}, \vec{V})}{\cos \phi_{\text{out}}}. \qquad (3.38)$$
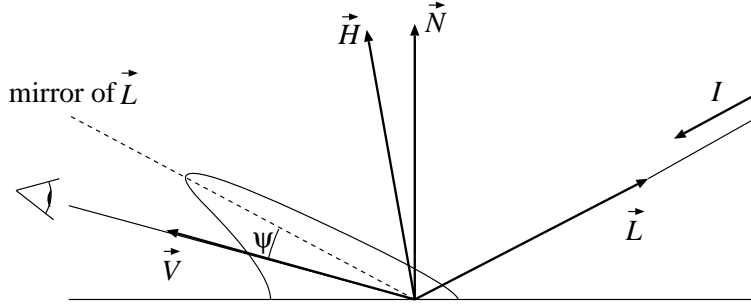
Figure 3.10: Specular reflection

Since $R_s(\vec{L}, \vec{V})$ is relevant only when $\vec{V}$ is close to the mirror direction of $\vec{L}$:

$$\cos \phi_{\text{in}} \cdot \frac{R_s(\vec{L}, \vec{V})}{\cos \phi_{\text{out}}} \approx R_s(\vec{L}, \vec{V}). \qquad (3.39)$$

To describe the intensity peak mathematically, a bi-directional function had to be proposed, which is relatively smooth, easy to control and simple to compute. Phong used the $k_s \cdot \cos^n \psi$ function for this purpose, where ψ is the angle between the direction of interest and the mirror direction, n is a constant describing how shiny the surface is, and k_s is the **specular coefficient** representing the fraction of the specular reflection in the total reflected light.

Comparing this model to real world measurements we can conclude that the specular coefficient k_s does not depend on the object's "own color" (in the highlights we can see the color of the lightsource rather than the color of the object), but that it does depend on the angle between the mirror direction and the surface normal, as we shall see in the next section.

The simplified illumination formula is then:

$$I^{\text{out}}(\lambda) = I^{\text{in}}(\lambda) \cdot k_d(\lambda) \cdot \cos \phi_{\text{in}} + I^{\text{in}}(\lambda) \cdot k_s(\lambda, \phi_{\text{in}}) \cdot \cos^n \psi. \qquad (3.40)$$

Let the **halfway unit vector** of $\vec{L}$ and $\vec{V}$ be $\vec{H} = (\vec{L} + \vec{V})/|\vec{L} + \vec{V}|$. The term $\cos \psi$ can be calculated from the dot product of unit vectors $\vec{V}$ and $\vec{H}$, since according to the law of reflection:

$$\psi = 2 \cdot \text{angle}(\vec{N}, \vec{H}). \qquad (3.41)$$

By trigonometric considerations:

$$\cos\psi = \cos(2 \cdot \text{angle}(\vec{N}, \vec{H})) = 2 \cdot \cos^2(\text{angle}(\vec{N}, \vec{H})) - 1 = 2 \cdot (\vec{N} \cdot \vec{H})^2 - 1 \tag{3.42}$$

Should the result turn out to be a negative number, the observer and the lightsource are obviously on different sides of the surface, and thus the specular term is zero. If the surface is lit not only by a single collimated beam, the right side of this expression has to be integrated over the hemisphere, or if several collimated beams target the surface, their contribution should simply be added up. It is important to note that, unlike Lambert's law, this model has no physical interpretation, but it follows nature in an empirical way only.

3.5 Probabilistic modeling of specular reflection

Specular reflection can be more rigorously analyzed by modeling the surface irregularities by probability distributions, as has been proposed by Torrance, Sparrow, Cook and Blinn. In their model, the surface is assumed to consist of randomly oriented perfect mirrors, so-called **microfacets**. As in the previous section, the reflected light is broken down into diffuse and specular components. The diffuse component is believed to be generated by multiple reflections on the microfacets and also by emission of the absorbed light by the material of the surface. The diffuse component is well described by Lambert's law. The specular component, on the other hand, is produced by the direct reflections of the microfacets. The bi-directional reflection function is also broken down accordingly, and we will discuss the derivation of the specular bi-directional reflection function $R_s(\vec{L}, \vec{V})$:

$$R(\vec{L}, \vec{V}) = R_d(\vec{L}, \vec{V}) + R_s(\vec{L}, \vec{V}) = k_d \cdot \cos\phi_{\text{out}} + R_s(\vec{L}, \vec{V}). \tag{3.43}$$

Returning to the original definition, the bi-directional reflection function is, in fact, an additive component of a probability density function, which is true for R_s as well.

$$R_s(\vec{L}, \vec{V}) \cdot d\omega =$$

Pr{photon is reflected directly to $d\omega$ around $\vec{V}$ | coming from $\vec{L}$}. (3.44)

Concerning this type of reflection from direction $\vec{L}$ to $d\omega$ around direction $\vec{V}$, only those facets can contribute whose normal is in $d\omega_H$ around the halfway unit vector $\vec{H}$. If reflection is to happen, the facet must obviously be facing in the right direction. It should not be hidden by other facets, nor should its reflection run into other facets, and it should not absorb the photon for the possible contribution.
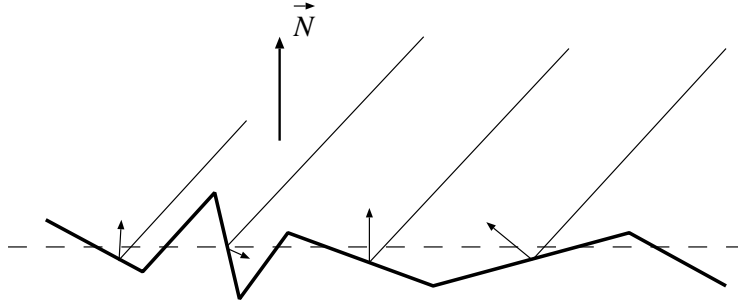

Figure 3.11: Microfacet model of the reflecting surface

Considering these facts, the event that "a photon is reflected directly to $d\omega$ around $\vec{V}$" can be expressed as the logical AND connection of the following stochastically independent events:

1. **Orientation**: In the path of the photon there is a microfacet having its normal in $d\omega_H$ around $\vec{H}$.

2. **No shadowing or masking**: The given microfacet is not hidden by other microfacets from the photon coming from the lightsource, and the reflected photon does not run into another microfacet.

3. **Reflection**: The photon is not absorbed by the perfect mirror.

Since these events are believed to be stochastically independent, their probability can be calculated independently, and the probability of the composed event will be their product.

Concerning the probability of the microfacet normal being in $d\omega_H$, we can suppose that all facets have equal area f. Let the probability density of the number of facets per unit area surface, per solid angle of facet normal

be $P(\vec{H})$. Blinn [Bli77] proposed **Gaussian distribution** for $P(\vec{H})$, since it seemed reasonable due to the central value theorem of probability theory:

$$P(\vec{H}) = \text{const} \cdot e^{-(\alpha/m)^2}. \tag{3.45}$$

where α is the angle of the microfacet with respect to the normal of the mean surface, that is the angle between $\vec{N}$ and $\vec{H}$, and m is the root mean square of the slope, i.e. a measure of the roughness.

Later Torrance and Sparrow showed that the results of the early work of Beckmann [BS63] and Davies [Dav54], who discussed the scattering of electromagnetic waves theoretically, can also be used here and thus Torrance proposed the **Beckmann distribution** function instead of the Gaussian:

$$P(\vec{H}) = \frac{1}{m^2 \cos^4 \alpha} \cdot e^{-\left(\frac{\tan^2 \alpha}{m^2}\right)}. \tag{3.46}$$

If a photon arrives from direction $\vec{L}$ to a surface element dA, the visible area of the surface element will be $dA \cdot (\vec{N} \cdot \vec{L})$, while the total visible area of the microfacets having their normal in the direction around $\vec{H}$ will be

$$f \cdot P(\vec{H}) \cdot d\omega_H \cdot dA \cdot (\vec{H} \cdot \vec{L}).$$

The probability of finding an appropriate microfacet aligned with the photon can be worked out as follows:

$$\Pr\{\text{orientation}\} = \frac{f \cdot P(\vec{H}) \cdot d\omega_H \cdot dA \cdot (\vec{H} \cdot \vec{L})}{dA \cdot (\vec{N} \cdot \vec{L})} = \frac{f \cdot P(\vec{H}) \cdot d\omega_H \cdot (\vec{H} \cdot \vec{L})}{(\vec{N} \cdot \vec{L})}. \tag{3.47}$$

The visibility of the microfacets from direction $\vec{V}$ means that the reflected photon does not run into another microfacet. The collision is often referred to as masking. Looking at figure 3.12, we can easily recognize that the probability of masking is l_1/l_2, where l_2 is the one-dimensional length of the microfacet, and l_1 describes the boundary case from where the beam is masked. The angles of the triangle formed by the bottom of the microfacet wedge and the beam in the boundary case can be expressed by the angles $\alpha = \text{angle}(\vec{N}, \vec{H})$ and $\beta = \text{angle}(\vec{V}, \vec{H}) = \text{angle}(\vec{L}, \vec{H})$ by geometric considerations and by using the law of reflection. Applying the sine law for this triangle, and some trigonometric formulae:

$$\Pr\{\text{not masking}\} = 1 - \frac{l_1}{l_2} = 1 - \frac{\sin(\beta + 2\alpha - \pi/2)}{\sin(\pi/2 - \beta)} = 2 \cdot \frac{\cos\alpha \cdot \cos(\beta + \alpha)}{\cos\beta}. \tag{3.48}$$
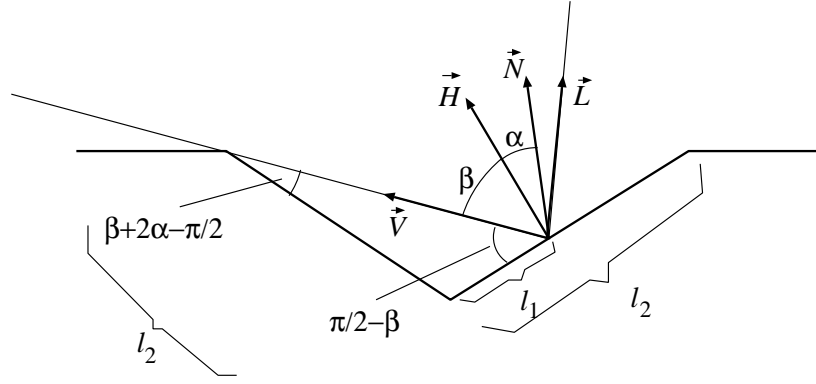
Figure 3.12: Geometry of masking

According to the definitions of the angles $\cos \alpha = \vec{N} \cdot \vec{H}$, $\cos(\beta + \alpha) = \vec{N} \cdot \vec{V}$ and $\cos \beta = \vec{V} \cdot \vec{H}$.

If the angle of incident light and the facet normal do not allow the triangle to be formed, the probability of no masking taking place is obviously 1. This situation can be recognized by evaluating the formula without any previous considerations and checking whether the result is greater than 1, then limiting the result to 1. The final result is:

$$\Pr\{\text{not masking}\} = \min\{2 \cdot \frac{(\vec{N} \cdot \vec{H}) \cdot (\vec{N} \cdot \vec{V})}{(\vec{V} \cdot \vec{H})}, 1\}. \qquad (3.49)$$

The probability of shadowing can be derived in exactly the same way, only $\vec{L}$ should be substituted for $\vec{V}$:

$$\Pr\{\text{not shadowing}\} = \min\{2 \cdot \frac{(\vec{N} \cdot \vec{H}) \cdot (\vec{N} \cdot \vec{L})}{(\vec{L} \cdot \vec{H})}, 1\}. \qquad (3.50)$$

The probability of neither shadowing nor masking taking place can be approximated by the minimum of the two probabilities:

$$\Pr\{\text{no shadow and mask}\} \approx$$

$$\min\{2 \cdot \frac{(\vec{N} \cdot \vec{H}) \cdot (\vec{N} \cdot \vec{V})}{(\vec{V} \cdot \vec{H})}, 2 \cdot \frac{(\vec{N} \cdot \vec{H}) \cdot (\vec{N} \cdot \vec{L})}{(\vec{L} \cdot \vec{H})}, 1\} = G(\vec{N}, \vec{L}, \vec{V}). \quad (3.51)$$
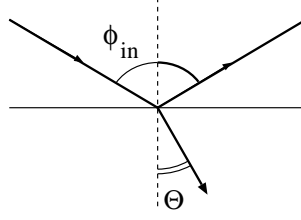
Figure 3.13: Reflection and refraction of a surface

Even perfect mirrors absorb some portion of the incident light, as is described by the **Fresnel equations** of physical optics, expressing the reflection (F) of a perfectly smooth mirror in terms of the refractive index of the material, ν, the extinction coefficient, κ which describes the conductivity of the material (for nonmetals $\kappa = 0$), and the angle of the incidence of the light beam, ϕ_{in}. Using the notations of figure 3.13, where ϕ_{in} is the incident angle and θ is the angle of refraction, the Fresnel equation expressing the ratio of the energy of the reflected beam and the energy of the incident beam for the directions parallel and perpendicular to the electric field is:

$$F_{\perp}(\lambda, \phi_{\text{in}}) = |\frac{\cos\theta - (\nu + \kappa\jmath) \cdot \cos\phi_{\text{in}}}{\cos\theta + (\nu + \kappa\jmath) \cdot \cos\phi_{\text{in}}}|^2, \tag{3.52}$$

$$F_{\parallel}(\lambda, \phi_{\text{in}}) = |\frac{\cos\phi_{\text{in}} - (\nu + \kappa\jmath) \cdot \cos\theta}{\cos\phi_{\text{in}} + (\nu + \kappa\jmath) \cdot \cos\theta}|^2, \tag{3.53}$$

where $\jmath = \sqrt{-1}$. These equations can be derived from Maxwell's fundamental formulae describing the basic laws of electric waves. If the light is unpolarized, that is, the parallel ($\vec{E}_{\parallel}$) and the perpendicular ($E_{\perp}$) electric fields have the same amplitude, the total reflectivity is:

$$F(\lambda, \phi_{\text{in}}) = \frac{|F_{\parallel}^{1/2} \cdot \vec{E}_{\parallel} + F_{\perp}^{1/2} \cdot \vec{E}_{\perp}|^2}{|\vec{E}_{\parallel} + \vec{E}_{\perp}|^2} = \frac{F_{\parallel} + F_{\perp}}{2}. \tag{3.54}$$

Note that F is wavelength dependent, since n and κ are functions of the wavelength.

Parameters ν and κ are often not available, so they should be estimated from measurements, or from the value of the normal reflection if the extinction is small. At normal incidence ($\phi_{\text{in}} = 0$), the reflection is:

$$F_0(\lambda) = |\frac{1 - (\nu + \kappa j)}{1 + (\nu + \kappa j)}|^2 = \frac{(\nu - 1)^2 + \kappa^2}{(\nu + 1)^2 + \kappa^2} \approx [\frac{\nu - 1}{\nu + 1}]^2. \qquad (3.55)$$

Solving for ν gives the following equation:

$$\nu(\lambda) = \frac{1 + \sqrt{F_0(\lambda)}}{1 - \sqrt{F_0(\lambda)}}. \qquad (3.56)$$

F_0 can easily be measured, thus this simple formula is used to compute the values of the index of refraction ν. Values of $\nu(\lambda)$ can then be substituted into the Fresnel equations (3.52 and 3.53) to obtain reflection parameter F for other angles of incidence.

Since F is the fraction of the reflected energy, it also describes the probability of a photon being reflected, giving:

$$\text{Pr}\{\text{reflection}\} = F(\lambda, \vec{N} \cdot \vec{L})$$

where variable ϕ_{in} has been replaced by $\vec{N} \cdot \vec{L}$.

Now we can summarize the results by multiplying the probabilities of the independent events to express $R_s(\vec{L}, \vec{V})$:

$$R_s(\vec{L}, \vec{V}) = \frac{1}{d\omega} \text{Pr}\{\text{orientation}\} \cdot \text{Pr}\{\text{no mask and shadow}\} \cdot \text{Pr}\{\text{reflection}\} =$$

$$\frac{d\omega_H}{d\omega} \frac{f \cdot P(\vec{H}) \cdot (\vec{H} \cdot \vec{L})}{(\vec{N} \cdot \vec{L})} \cdot G(\vec{N}, \vec{L}, \vec{V}) \cdot F(\lambda, \vec{N} \cdot \vec{L}). \qquad (3.57)$$

The last problem left is the determination of $d\omega_H/d\omega$ [JGMHe88]. Defining a spherical coordinate system (ϕ, θ), with the north pole in the direction of $\vec{L}$ (figure 3.14), the solid angles are expressed by the product of vertical and horizontal arcs:

$$d\omega = dV_{\text{hor}} \cdot dV_{\text{vert}}, \qquad d\omega_H = dH_{\text{hor}} \cdot dH_{\text{vert}}. \qquad (3.58)$$
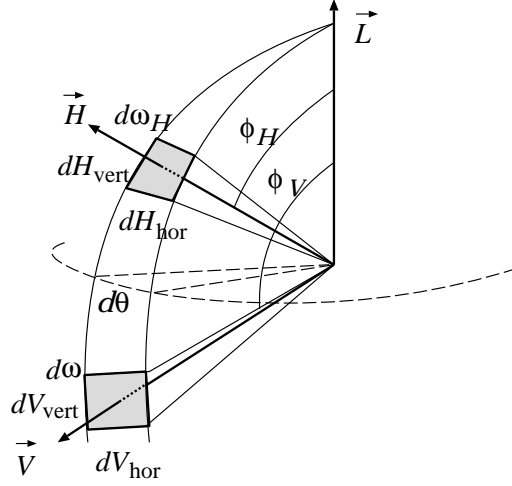
Figure 3.14: Calculation of $d\omega_H/d\omega$

By using geometric considerations and applying the law of reflection, we get:

$$dV_{\text{hor}} = d\theta \cdot \sin \phi_V, \qquad dH_{\text{hor}} = d\theta \cdot \sin \phi_H, \qquad dV_{\text{vert}} = 2dH_{\text{vert}}. \quad (3.59)$$

This in turn yields:

$$\frac{d\omega_H}{d\omega} = \frac{\sin \phi_H}{2\sin \phi_V} = \frac{\sin \phi_H}{2\sin 2\phi_H} = \frac{1}{4\cos \phi_H} = \frac{1}{4(\vec{L} \cdot \vec{H})}. \quad (3.60)$$

since $\phi_V = 2 \cdot \phi_H$.

The final form of the specular term is:

$$R_s(\vec{L}, \vec{V}) = \frac{f \cdot P(\vec{H})}{4(\vec{N} \cdot \vec{L})} \cdot G(\vec{N}, \vec{L}, \vec{V}) \cdot F(\lambda, \vec{N} \cdot \vec{L}). \quad (3.61)$$

3.6 Abstract lightsource models

Up to now we have dealt with lightsources as ordinary surfaces with positive emission I_e. Simplified illumination models, however, often make a distinction between "normal" surfaces, and some abstract objects called **"lightsources"**. These abstract lightsources cannot be seen on the image directly, they are only responsible for feeding energy into the system and thus making the normal surfaces visible for the camera.

The most common types of such lightsources are the following:

1. **Ambient light** is assumed to be constant in all directions and to be present everywhere in 3D space. Its role in restoring the energy equilibrium is highlighted in the next section.

2. **Directional lightsources** are supposed to be at infinity. Thus the light beams coming from a directional lightsource are parallel and their energy does not depend on the position of the surface. (The sun behaves like a directional lightsource.)

3. **Positional or point lightsources** are located at a given point in the 3D space and are concentrated on a single point. The intensity of the light at distance d is $I_l(d) = I_0 \cdot f(d)$. If it really were a point-like source, $f(d) = 1/d^2$ should hold, but to avoid numerical instability for small distances, we use $f(d) = 1/(a \cdot d + b)$ instead, or to emphasize atmospheric effects, such as fog $f(d) = 1/(a \cdot d^m + b)$ might also be useful (m, a and b are constants).

4. **Flood lightsources** are basically positional ligthsources with radiant intensity varying with the direction of interest. They have a main radiant direction, and as the angle (ξ) of the main and actual directions increases the intensity decreases significantly. As for the Phong model, the function $\cos^n \xi$ seems appropriate:

$$I_l(d, \xi) = I_0 \cdot f(d) \cdot \cos^n \xi. \qquad (3.62)$$

These abstract lightsources have some attractive properties which ease color computations. Concerning a point on a surface, an abstract lightsource may only generate a collimated beam onto the surface point, with

the exception of ambient light. This means that the integral of the rendering equation can be simplified to a summation with respect to different lightsources, if the indirect light reflected from other surfaces is ignored. The direction of the collimated beam can also be easily derived, since for a directional lightsource it is a constant parameter of the lightsource itself, and for positional and flood lightsources it is the vector pointing from the point-like lightsource to the surface point. The reflection of ambient light, however, can be expressed in closed form, since only a constant function has to be integrated over the hemisphere.

3.7 Steps for image synthesis

The final objective of graphics algorithms is the calculation of pixel colors, or their respective (R, G, B) values. According to our model of the camera (or eye), this color is defined by the energy flux leaving the surface of the visible object and passing through the pixel area of the window towards the camera. As has been proven in the previous section, this flux is proportional to the intensity of the surface in the direction of the camera and the projected area of the pixel, and is independent of the distance of the surface if it is finite (equation 3.19).

Intensity $I(\lambda)$ has to be evaluated for that surface which is visible through the given pixel, that is the nearest of the surfaces located along the line from the camera towards the center of the pixel. The determination of this surface is called the **hidden surface problem** or **visibility calculation** (chapter 6). The computation required by the visibility calculation is highly dependent on the coordinate system used to specify the surfaces, the camera and the window. That makes it worth transforming the virtual world to a coordinate system fixed to the camera, where this calculation can be more efficient. This step is called **viewing transformation** (chapter 5). Even in a carefully selected coordinate system, the visibility calculation can be time-consuming if there are many surfaces, so it is often carried out after a preprocessing step, called **clipping** (section 5.5) which eliminates those surface parts which cannot be projected onto the window.

Having solved the visibility problem, the surface visible in the given pixel is known, and the radiant intensity may be calculated on the representative wavelengths by the following **shading** equation (the terms of diffuse and

specular reflections have already been substituted into equation 3.30):

$$I^{\mathrm{out}} = I_e + k_r \cdot I_r + k_t \cdot I_t + \int\limits^{2\pi} k_d \cdot I^{\mathrm{in}} \cdot \cos \phi_{\mathrm{in}} \, d\omega_{\mathrm{in}} + \int\limits^{2\pi} k_s \cdot I^{\mathrm{in}} \cdot \cos^n \psi \, d\omega_{\mathrm{in}}. \quad (3.63)$$
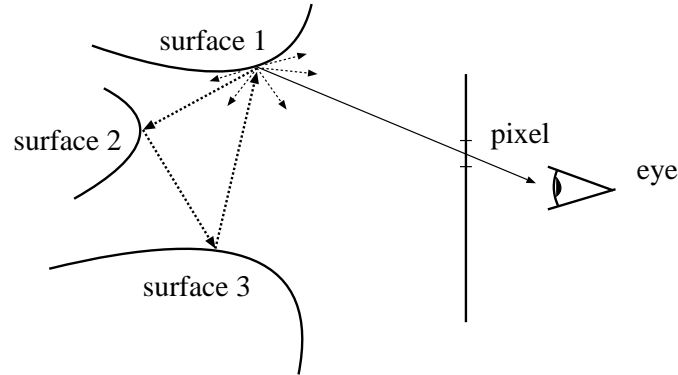

Figure 3.15: Multiple reflections of light

Due to multiple reflections of light beams, the calculation of the intensity of the light leaving a point on a surface in a given direction requires the intensities of other surfaces visible from this point, which of course generates new visibility and **shading** problems to solve (figure 3.15). It must be emphasized that these other surfaces are not necessarily inside the clipping region. To calculate those intensities, other surfaces should be evaluated, and our original point on the given surface might contribute to those intensities. As a consequence of that, the formula has complicated coupling between its left and right sides, making the evaluation difficult.

There are three general and widely accepted approaches to solve this integral equation in 3D continuous space.

1. **The analytical solution**

   Analytical methods rely on numerical techniques to solve the integral equation in its original or simplified form. One of the most popular numerical techniques is the finite element method. Its first step is the subdivision of the continuous surfaces into elemental surface patches, making it possible to approximate their intensity distribution

by constant values independent of the position on the surface. Taking advantage of this homogeneous property of the elemental patches, a stepwise constant function should be integrated in our formula, which can be substituted by a weighted sum of the unknown patch parameters. This step transforms the integral equation into a linear equation which can be solved by straightforward methods. It must be admitted that this solution is not at all simple because of the size of the resulting linear system, but at least we can turn to the rich toolset of the numerical methods of linear algebra. The application of the analytical approach to solve the integral equation of the shading formula in 3D space leads to the family of **analytical shading models**, or as it is usually called, the **radiosity method**.

2. **Constraining the possible coupling**

   Another alternative is to eliminate from the rendering equation those energy contributions which cause the difficulties, and thus give ourselves a simpler problem to solve. For example, if coherent coupling of limited depth, say n, were allowed, and we were to ignore the incoherent component coming from non-abstract lightsources, then the number of surface points which would need to be evaluated to calculate a pixel color can be kept under control. Since the illumination formula contains two terms regarding the coherent components (reflective and refracting lights), the maximum number of surfaces involved in the color calculation of a pixel is two to the power of the given depth, i.e. 2^n. The implementation of this approach is called **recursive ray tracing**.

3. **Ignoring the coupling**

   An even more drastic approach is to simplify or disregard completely all the terms causing problems, and to take into account only the incoherent reflection of the light coming from the abstract lightsources, and the coherent transmission supposing the index of refraction to be equal to 1. Mirrors and refracting objects cannot be described in this model. Since the method is particularly efficient when used with incremental hidden surface methods and can also be implemented using the incremental concept, it is called the **incremental shading method**.

The three different approaches represent three levels of the compromise between image generation speed and quality. By ignoring more and more terms in the illumination formula, its calculation can be speeded up, but the result inevitably becomes more and more artificial. The shading methods based on radiosity and ray tracing techniques or on the combination of the two form the family of **photorealistic image generation**. Simple, incremental shading algorithms, on the other hand, are suitable for very fast hardware implementation, making it possible to generate real-time animated sequences.

The simplification of the illumination formula has been achieved by ignoring some of its difficult-to-calculate terms. Doing this, however, violates the energy equilibrium, and causes portions of objects to come out extremely dark, sometimes unexpectedly so. These artifacts can be reduced by reintroducing the ignored terms in simplified form, called **ambient light**. The ambient light represents the ignored energy contribution in such a way as to satisfy the energy equilibrium. Since this ignored part is not calculated, nothing can be said of its positional and directional variation, hence it is supposed to be constant in all directions and everywhere in the 3D space. From this aspect, the role of ambient light also shows the quality of the shading algorithm. The more important a role it has, the poorer quality picture it will generate.

Chapter 4

MODEL DECOMPOSITION

The term **model decomposition** refers to the operation when the database describing an object scene is processed in order to produce simple geometric entities which are suitable for image synthesis. The question of which sorts of geometric entity are suitable for picture generation can be answered only if one is aware of the nature of the image synthesis algorithm to be used. Usually these algorithms cannot operate directly with the world representation. The only important exception is the ray tracing method (see chapter 9 and section 6.1 in chapter 6) which works with practically all types of representation scheme. Other types of image synthesis algorithm, however, require special types of geometric entity as their input. These geometric entities are very simple and are called **graphics primitives**. Thus model decomposition produces low level graphics primitives from a higher level representation scheme. Usually these primitives are polygons or simply triangles. Since many algorithms require triangles only as their input, and polygons can be handled similarly, this chapter will examine that case of model decomposition in which the graphics primitives are triangles. The problem is the following: a solid object given by a representation scheme, approximate its boundary by a set of triangles.

The most straightforward approach to this task is to generate a number of surface points so that they can be taken as triangle vertices. Each triangle then becomes a linear interpolation of the surface between the three vertices.

The resulting set of triangles is a valid mesh if for each triangle:

- each of its vertices is one of the generated surface points

- each of its edges is shared by exactly one other (neighboring) triangle except for those that correspond to the boundary curve of the surface

- there is no other triangle which intersects it, except for neighboring triangles sharing common edges or vertices

Some image synthesis algorithms also require their input to contain topological information (references from the triangles to their neighbors); some do not, depending on the nature of the algorithm. It is generally true, however, that a consistent and redundancy-free mesh structure that stores each geometric entity once only (triangle vertices, for example, are not stored as many times as there are triangles that contain them) is usually much less cumbersome than a stack of triangles stored individually. For the sake of simplicity, however, we will concentrate here only on generating the triangles and omit topological relationships between them.

4.1   Simple geometric objects

A geometric object is usually considered to be *simple* if it can be described by one main formula characterizing its shape and (possibly) some additional formulae characterizing its actual boundary. In other words, a simple geometric object has a *uniform shape*. A sphere with center $c \in E^3$ and of radius r is a good example, because its points p satisfy the formula:

$$|p - c| \leq r \tag{4.1}$$

where $|\cdot|$ denotes vector length.

   Simple objects are also called *geometric primitives*. The task is to approximate the surface of a primitive by a triangular mesh, that is, a number of surface points must be generated and then proper triangles must be formed. In order to produce surface points, the formula describing the surface must have a special form called *explicit form*, as will soon become apparent.

4.1.1   Explicit surface patches

The formula describing a surface is in (biparametric) *explicit form* if it characterizes the coordinates (x, y, z) of the surface points in the following way:

$$
\begin{aligned}
x &= f_x(u, v), \\
y &= f_y(u, v), \\
z &= f_z(u, v), \qquad (u, v) \in D
\end{aligned}
\tag{4.2}
$$

where D is the 2D parameter domain (it is usually the rectangular box defined by the inequalities $0 \le u \le u_{\max}$ and $0 \le v \le v_{\max}$ for the most commonly used 4-sided patches). The formula "generates" a surface point at each parameter value (u, v), and the continuity of the functions f_x, f_y, f_z ensures that each surface point is generated at some parameter value (the formulae used in solid modeling are analytic or more often algebraic which implies continuity; see subsection 1.6.1). This is exactly what is required in model decomposition: the surface points can be generated (sampled) to any desired resolution.

In order to generate a valid triangular mesh, the 2D parameter domain, D, must be sampled and then proper triangles must be formed from the sample points. We distinguish between the following types of faces (patches) with respect to the shape of D.

Quadrilateral surface patches

The most commonly used form of the parameter domain D is a rectangular box in the parameter plane, defined by the following inequalities:

$$
0 \le u \le u_{\max}, \qquad 0 \le v \le v_{\max}.
\tag{4.3}
$$

The resulting patch is 4-sided in this case, and the four boundary curves correspond to the boundary edges of the parameter rectangle ($\cdot$ stands for any value of the domain): $(0, \cdot)$, $(u_{\max}, \cdot)$, $(\cdot, 0)$, $(\cdot, v_{\max})$. The curves defined by parameter ranges $(u, \cdot)$ or $(\cdot, v)$, that is, where one of the parameters is fixed, are called *isoparametric* curves. Let us consider the two sets of isoparametric curves defined by the following parameter ranges (the subdivision is not necessarily uniform):

$$
\begin{aligned}
&(0, \cdot), (u_1, \cdot), \quad \ldots, \quad (u_{n-1}, \cdot), (u_{\max}, \cdot), \\
&(\cdot, 0), (\cdot, v_1), \quad \ldots, \quad (\cdot, v_{m-1}), (\cdot, v_{\max}).
\end{aligned}
\tag{4.4}
$$

The two sets of curves form a quadrilateral mesh on the surface. The vertices of each quadrilateral correspond to parameter values of the form (u_i, v_j), (u_{i+1}, v_j), (u_{i+1}, v_{j+1}), (u_i, v_{j+1}). Each quadrilateral can easily be cut into two triangles and thus the surface patch can be approximated by $2nm$ number of triangles using the following simple algorithm (note that it realizes a uniform subdivision):

DecomposeQuad$(\vec{f}, n, m)$ // $\vec{f} = (f_x, f_y, f_z)$
    $S = \{\}$; $u_i = 0$;   // S: *resulting set of triangles*
    for $i = 1$ **to** n **do**
        $u_{i+1} = u_{\max} \cdot i/n$; $v_j = 0$;
        for $j = 1$ **to** m **do**
            $v_{j+1} = v_{\max} \cdot j/m$;
            add the triangle $\vec{f}(u_i, v_j)$, $\vec{f}(u_{i+1}, v_j)$, $\vec{f}(u_{i+1}, v_{j+1})$ to S;
            add the triangle $\vec{f}(u_i, v_j)$, $\vec{f}(u_{i+1}, v_{j+1})$, $\vec{f}(u_i, v_{j+1})$ to S;
            $v_j = v_{j+1}$;
        endfor
        $u_i = u_{i+1}$;
    endfor
    return S;
end
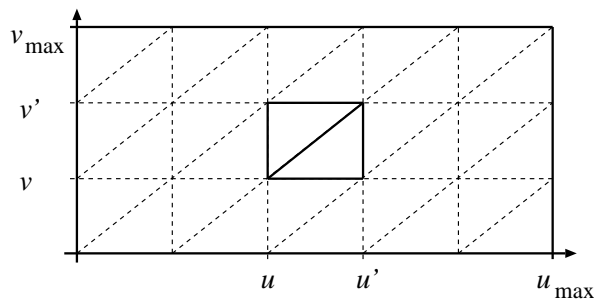
Figure 4.1: Subdivision of a rectangular parameter domain

Note that the quadrilateral (triangular) subdivision of the patch corresponds to a quadrilateral (triangular) subdivision of the parameter domain

D, as illustrated in figure 4.1. This is not surprising, since the mapping $f(u, v)$ is a continuous and one-to-one mapping, and as such, preserves topological invariances, for example neighborhood relationships.

Triangular surface patches

Triangular — and more generally; non-quadrilateral — surface patches were introduced into geometric modeling because the fixed topology of surfaces based on 4-sided patches restricted the designer's freedom in many cases (non-quadrilateral patches are typically necessary for modeling rounded corners where three or more other patches meet and must be blended). The parameter domain D is usually triangle-shaped. The Steiner patch [SA87], for example, is defined over the following parameter domain:

$$u \geq 0, \quad v \geq 0, \quad u + v \leq 1 \qquad (4.5)$$

It often occurs, however, that the triangular patch is parameterized via three parameters, that is having the form $f(u, v, w)$, but then the three parameters are not mutually independent. The Bezier triangle is an example of this (see any textbook on surfaces in computer aided geometric design, such as [Yam88]). Its parameter domain is defined as:

$$u \geq 0, \quad v \geq 0, \quad w \geq 0, \quad u + v + w = 1 \qquad (4.6)$$

It is also a triangle, but defined in a 3D coordinate system. In order to discuss the above two types of parameter domain in a unified way, the parameter will be handled as a vector $\vec{u}$ which is either a 2D or a 3D vector, that is a point of a 2D or 3D parameter space U. The parameter domain $D \subset U$ is then defined as a triangle spanned by the three vertices $\vec{u}_1, \vec{u}_2, \vec{u}_3 \in U$.

The task is to subdivide the triangular domain D into smaller triangles. Of all the imaginable variations on this theme, the neatest is perhaps the following, which is based on recursive subdivision of the triangle into similar smaller ones using the middle points of the triangle sides. As illustrated in figure 4.2, the three middle points, $\vec{m}_1, \vec{m}_2, \vec{m}_3$, are generated first:

$$\vec{m}_1 = \frac{1}{2}(\vec{u}_2 + \vec{u}_3), \quad \vec{m}_2 = \frac{1}{2}(\vec{u}_3 + \vec{u}_1), \quad \vec{m}_3 = \frac{1}{2}(\vec{u}_1 + \vec{u}_2). \qquad (4.7)$$
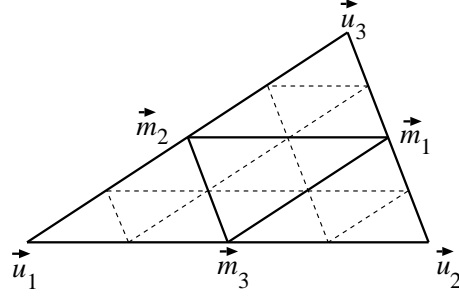
Figure 4.2: Subdivision of a triangular parameter domain

The resulting four smaller triangles are then further subdivided in a similar way. The subdivision continues until a predefined "depth of recurrence", say d, is reached. The corresponding recursive algorithm is the following:

DecomposeTriang($\vec{f}$, $\vec{u}_1$, $\vec{u}_2$, $\vec{u}_3$, d) $// \vec{f} = (f_x, f_y, f_z)$
    if $d \leq 0$ **then return** the triangle of vertices $\vec{f}(\vec{u}_1)$, $\vec{f}(\vec{u}_2)$, $\vec{f}(\vec{u}_3)$;
    $S = \{\}$;
    $\vec{m}_1 = \frac{1}{2}(\vec{u}_2 + \vec{u}_3)$; $\vec{m}_2 = \frac{1}{2}(\vec{u}_3 + \vec{u}_1)$; $\vec{m}_3 = \frac{1}{2}(\vec{u}_1 + \vec{u}_2)$;
    add **DecomposeTriang**($\vec{f}$, $\vec{u}_1$, $\vec{m}_3$, $\vec{m}_2$, $d-1$) to S;
    add **DecomposeTriang**($\vec{f}$, $\vec{u}_2$, $\vec{m}_1$, $\vec{m}_3$, $d-1$) to S;
    add **DecomposeTriang**($\vec{f}$, $\vec{u}_3$, $\vec{m}_2$, $\vec{m}_1$, $d-1$) to S;
    add **DecomposeTriang**($\vec{f}$, $\vec{m}_1$, $\vec{m}_2$, $\vec{m}_3$, $d-1$) to S;
    return S;
end

General n-sided surface patches

Surface patches suitable for interpolating curve networks with general (irregular) topology are one of the most recent achievements in geometric modeling (see [Vár87] or [HRV92] for a survey). The parameter domain corresponding to an n-sided patch is usually an n-sided convex polygon (or even a regular n-sided polygon with sides of unit length as in the case of

the so-called overlap patches [Vár91]). A convex polygon can easily be broken down into triangles, as will be shown in subsection 4.2.1, and then the triangles can be further divided into smaller ones.

4.1.2 Implicit surface patches

The formula describing a surface is said to be in *implicit form* if it characterizes the coordinates (x, y, z) of the surface points in the following way:

$$f(x, y, z) = 0. \tag{4.8}$$

This form is especially suitable for tests that decide whether a given point is on the surface: the coordinates of the point are simply substituted and the value of f gives the result. Model decomposition, however, yields something of a contrary problem: points which are on the surface must be generated. The implicit equation does not give any help in this, it allows us only to check whether a given point does in fact lie on the surface. As we have seen in the previous subsection, explicit forms are much more suitable for model decomposition than implicit forms. We can conclude without doubt that the implicit form in itself is not suitable for model decomposition.

Two ways of avoiding the problems arising from the implicit form seem to exist. These are the following:

1. *Avoiding model decomposition.* It has been mentioned that ray tracing is an image synthesis method that can operate directly on the world representation. The only operation that ray tracing performs on the geometric database is the calculation of the intersection point between a light ray (directed semi-line) and the surface of an object. In addition, this calculation is easier to perform if the surface formula is given in implicit form (see subsection 6.1.2 about intersection with implicit surfaces).

2. *Explicitization.* One can try to find an explicit form which is equivalent to the given implicit form, that is, which characterizes the same surface. No general method is known, however, for solving the explicitization problem. The desired formulae can be obtained heuristically. Explicit formulae for simple surfaces, such as sphere or cylinder surfaces, can easily be constructed (examples can be found in subsection

12.1.2, where the problem is examined within the context of texture mapping).

The conclusion is that implicit surfaces are generally not suited to being broken down into triangular meshes, except for simple types, but this problem can be avoided by selecting an image synthesis algorithm (ray tracing) which does not require preliminary model decomposition.

4.2　Compound objects

Compound objects are created via special operations performed on simpler objects. The simpler objects themselves can also be compound objects, but the bottom level of this hierarchy always contains geometric primitives only. The operations by which compound objects can be created usually belong to one of the following two types:

1. *Regularized Boolean set operations.* We met these in subsection 1.6.1 on the general aspects of geometric modeling. Set operations are typically used in CSG representation schemes.

2. *Euler operators.* These are special operations that modify the boundary of a solid so that its combinatorial (topological) validity is left unchanged. The name relates to Euler's famous formula which states that the alternating sum of the number of vertices, edges and faces of a simply connected polyhedron is always two. This formula was then extended to more general polyhedra by geometric modelers. The Euler operators — which can create or remove vertices, edges and faces — are defined in such a way that performing them does not violate the formula [Män88], [FvDFH90]. Euler operators are typically used in B-rep schemes.

Although often not just one of the two most important representation schemes, CSG and B-rep, is used exclusively, that is, practical modeling systems use instead a hybrid representation, it is worth discussing the two schemes separately here.

4.2.1   Decomposing B-rep schemes

Breaking down a B-rep scheme into a triangular mesh is relatively simple. The faces of the objects are well described in B-rep, that is, not only their shapes but also their boundary edges and vertices are usually explicitly represented.
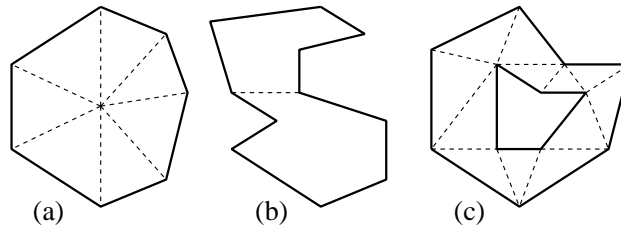

Figure 4.3: Polygon decompositions

If the object is a planar polyhedron, that is if it contains planar faces only, then each face can be individually retrieved and triangulated. Once the polygon has been broken down into triangles, generating a finer subdivision poses no real problem, since each triangle can be divided separately by the algorithm for triangular patches given in subsection 4.1.1. However the crucial question is: how to decompose a polygon — which is generally either convex or concave and may contain holes (that is multiply connected) — into triangles that perfectly cover its area and only its area. This *polygon triangulation problem*, like many others arising in computer graphics, has been studied in computational geometry. Without going into detail, let us distinguish between the following three cases (n denotes the number of vertices of the polygon):

1. *Convex polygons.* A convex polygon can easily be triangulated, as illustrated in part (a) of figure 4.3. First an inner point is calculated — for example the center of mass — and then each side of the polygon makes a triangle with this inner point. The time complexity of this operation is $O(n)$.

2. *Concave polygons without holes.* Such polygons cannot be triangulated in the previous way. A problem solving approach called **divide-and-conquer** can be utilized here in the following way. First two vertices of

the polygon must be found so that the straight line segment connecting them cuts the polygon into two parts (see part (b) of figure 4.3). This diagonal is called a *separator*. Each of the two resulting polygons is either a triangle, in which case it need not be divided further, or else has more than three vertices, so it can be divided further in a similar way. If it can be ensured that the two resulting polygons are of the same size (up to a ratio of two) with respect to the number of their vertices at each subdivision step, then this *balanced* recurrence results in a very good, $O(n \log n)$, time complexity. (Consult [Cha82] to see that the above property of the separator can always be ensured in not more than $O(n)$ time.)

3. *General polygons.* Polygons of this type may contain holes. A general method of triangulating a polygon with holes is to generate a *constrained triangulation* of its vertices, as illustrated in part (c) of figure 4.3. A triangulation of a set of points is an aggregate of triangles, where the vertices of the triangles are from the point set, no triangles overlap and they completely cover the convex hull of the point set. A triangulation is constrained if there are some predefined edges (point pairs) that must be triangle edges in the triangulation. Now the point set is the set of vertices and the constrained edges are the polygon edges. Having computed the triangulation, only those triangles which are inside the face need be retained. (Seidel [Sei88] shows, for example, how such a triangulation can be computed in $O(n \log n)$ time.)

Finally, if the object has curved faces, then their shape is usually described by (or their representation can be transformed to) explicit formulae. Since the faces of a compound object are the result of operations on primitive face elements (patches), and since usually their boundaries are curves resulting from intersections between surfaces, it cannot be assumed that the parameter domain corresponding to the face is anything as simple as a square or a triangle. It is generally a territory with a curved boundary, which can, however, be approximated by a polygon to within some desired tolerance. Having triangulated the original face the triangular faces can then be decomposed further until the approximation is sufficiently close to the original face.

4.2.2 Boundary evaluation for CSG schemes

As described in section 1.6.2 CSG schemes do not explicitly contain the faces of the objects, shapes are produced by combining half-spaces or primitives defining point sets in space. The boundary of the solid is *unevaluated* in such a representation. The operation that produces the faces of a solid represented by a CSG-tree is called **boundary evaluation**.

Set membership classification: the unified approach

Tilove has pointed out [Til80] that a paradigm called **set membership classification** can be a unified approach to geometric intersection problems arising in constructive solid geometry and related fields such as computer graphics. The classification of a *candidate set* X with respect to a *reference set* S maps X into the following three disjoint sets:

$$
\begin{array}{rcl}
C_{\mathrm{in}}(X,S) &=& X \cap iS, \\
C_{\mathrm{out}}(X,S) &=& X \cap cS, \\
C_{\mathrm{on}}(X,S) &=& X \cap bS,
\end{array}
\tag{4.9}
$$

where iS, cS, bS are the interior, complement and boundary of S, respectively. Note that if X is the union of boundaries of the primitive objects in the CSG-tree, and S is the solid represented by the tree, then boundary evaluation is no else but the computation of $C_{\mathrm{on}}(X,S)$. The exact computation of this set, however, will not be demonstrated here. An approximation method will be shown instead, which blindly generates all the patches that *may* fall onto the boundary of S and then tests each one whether to keep it or not. For this reason, the following binary *relations* can be defined between a candidate set X and a reference set S:

$$
\begin{array}{lll}
X \text{ in } S & \text{if} & X \subseteq iS, \\
X \text{ out } S & \text{if} & X \subseteq cS, \\
X \text{ on } S & \text{if} & X \subseteq bS
\end{array}
\tag{4.10}
$$

(note that either one or none of these can be true at a time for a pair X, S).

In constructive solid geometry, S is either a primitive object or is of the form $S = A \circ^* B$, where $\circ^*$ is one of the operations $\cup^*, \cap^*, \setminus^*$. A **divide-and-conquer** approach can now help us to simplify the problem.

The following relations are straightforward results:

$$
\begin{array}{lll}
X \text{ in } (A \cup B) & \text{if} & X \text{ in } A \;\vee\; X \text{ in } B \\
X \text{ out } (A \cup B) & \text{if} & X \text{ out } A \;\wedge\; X \text{ out } B \\
X \text{ on } (A \cup B) & \text{if} & (X \text{ on } A \;\wedge\; \neg\, X \text{ in } B) \;\vee\; (X \text{ on } B \;\wedge\; \neg\, X \text{ in } A)
\end{array}
$$

$$
\begin{array}{lll}
X \text{ in } (A \cap B) & \text{if} & X \text{ in } A \;\wedge\; X \text{ in } B \\
X \text{ out } (A \cap B) & \text{if} & X \text{ out } A \;\vee\; X \text{ out } B \\
X \text{ on } (A \cap B) & \text{if} & (X \text{ on } A \;\wedge\; \neg\, X \text{ in } cB) \;\vee\; (X \text{ on } B \;\wedge\; \neg\, X \text{ in } cA)
\end{array}
$$

$$
\begin{array}{lll}
X \text{ in } (A \setminus B) & \text{if} & X \text{ in } A \;\wedge\; X \text{ in } cB \\
X \text{ out } (A \setminus B) & \text{if} & X \text{ out } A \;\vee\; X \text{ out } cB \\
X \text{ on } (A \setminus B) & \text{if} & (X \text{ on } A \;\wedge\; \neg\, X \text{ in } B).
\end{array}
$$

$$(4.11)$$

That is, the classification with respect to a compound object of the form $S = A \circ B$ can be traced back to simple logical combinations of classification results with respect to the argument objects A, B.
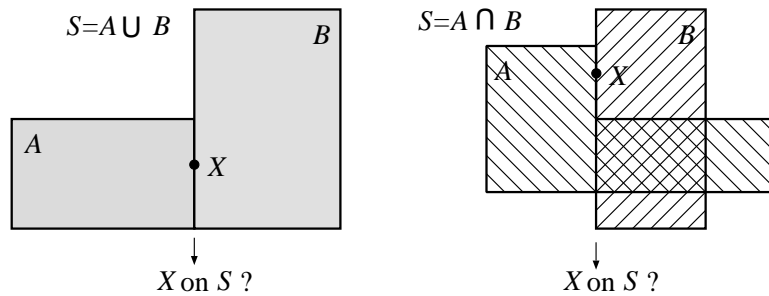
Figure 4.4: Some problematic situations in set membership classification

There are two problems, however:

1. The events on the right hand side are not equivalent with the events on the left hand side. The event X in $(A \cup B)$ can happen if X in A or X in B or (this is not contained by the expression) X_1 in A and X_2 in B where $X_1 \cup X_2 = X$. This latter event, however, is much more difficult to detect than the previous two.
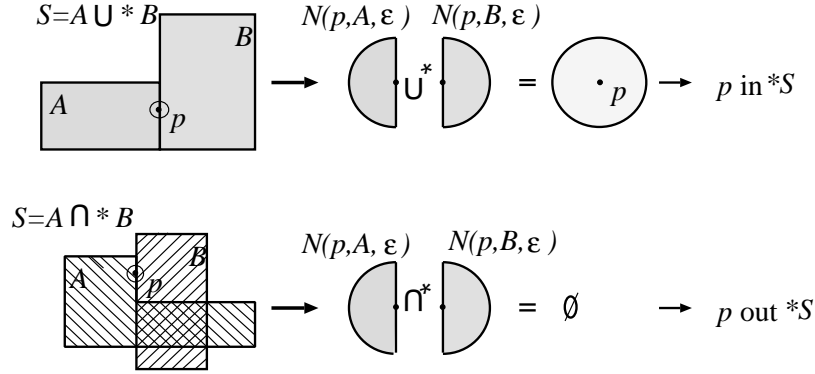
Figure 4.5: Regularizing set membership classifications

2. There are problematic cases when the above expressions are extended
   to regular sets and regularized set operations. Figure 4.4 illustrates
   two such problematic situations (the candidate set X is a single point
   in both cases).

Problem 1 can be overridden by a nice combination of a generate-and-test
and a divide-and-conquer strategy, as will soon be shown in the subsequent
sections.

The perfect *theoretical* solution to problem 2 is that each point $p \in X$ of
the candidate set is examined by considering a (sufficiently small) neighbor-
hood of p. Let us first consider the idea without worrying about implemen-
tation difficulties. Let $B(p, \varepsilon)$ denote a ball around p with a (small) radius
ε, and let $N(p, S, \varepsilon)$ be defined as the ε-neighborhood of p in S (see figure
4.5):

$$N(p, S, \varepsilon) = B(p, \varepsilon) \cap^* S. \tag{4.12}$$

Then the regularized set membership relations are the following:

$$\begin{aligned}
p \, \text{in}^* (A \circ^* B) \quad &\text{if} \quad \exists \varepsilon > 0: \ N(p, A, \varepsilon) \circ^* N(p, B, \varepsilon) = B(p, \varepsilon), \\
p \, \text{out}^* (A \circ^* B) \quad &\text{if} \quad \exists \varepsilon > 0: \ N(p, A, \varepsilon) \circ^* N(p, B, \varepsilon) = \emptyset, \\
p \, \text{on}^* (A \circ^* B) \quad &\text{if} \quad \forall \varepsilon > 0: \ \emptyset \neq N(p, A, \varepsilon) \circ^* N(p, B, \varepsilon) \neq B(p, \varepsilon).
\end{aligned} \tag{4.13}$$

Figure 4.5 shows some typical situations. One might suspect disappointing
computational difficulties in actually performing the above tests:

1. It is impossible to examine each point of a point set since their number is generally infinite. Intersection between point sets can be well computed by first checking whether one contains the other and if not, then intersecting their boundaries. If the point sets are polyhedra (as in the method to be introduced in the next part), then intersecting their boundaries requires simple computations (face/face, edge/face). Ensuring regularity implies careful handling of degenerate cases.

2. If a single point is to be classified then the ball of radius ε can, however, be substituted by a simple line segment of length 2ε and then the same operations performed on that as were performed on the ball in the above formulae. One must ensure then that ε is small enough, and that the line segment has a "general" orientation, that is, if it intersects the boundary of an object then the angle between the segment and tangent plane at the intersection point must be large enough to avoid problems arising from numerical inaccuracy of floating point calculations.

The conclusion is that the practical implementation of regularization does not mean the perfect imitation of the theoretical solution, but rather that simplified solutions are used and degeneracies are handled by keeping the theory in mind.

Generate-and-test

Beacon *et al.* [BDH$^+$89] proposed the following algorithm which *approximates* the boundary of a CSG solid, that is, generates surface patches the aggregate of which makes the boundary of the solid "almost perfectly" within a predefined tolerance. The basic idea is that the union of the boundaries of the primitive objects is a superset of the boundary of the compound solid, since each boundary point lies on some primitive. The approach based on this idea is a **generate-and-test** strategy:

1. The boundary of each primitive is roughly subdivided into patches in a preliminary phase and put onto a list L called the *candidate list*.

2. The patches on L are taken one by one and each patch P is classified with respect to the solid S, that is, the relations $P\operatorname{in}^* S$, $P\operatorname{out}^* S$ and $P\operatorname{on}^* S$ are evaluated.

3. If P on$^*S$ then P is put onto a list B called the *definitive boundary list*. This list will contain the result. If P in$^*S$ or P out$^*S$ then P is discarded since it cannot contribute to the boundary of S.

4. Finally, if none of the three relations holds, then P intersects the boundary of S somewhere, although it is not contained totally by it. In this case P should not be discarded but rather it is subdivided into smaller patches, say $P_1, \ldots, P_n$, which are put back onto the candidate list L. If, however, the size of P is below the predefined tolerance, then it is not subdivided further but placed onto a third list T called the *tentative boundary list*.

5. The process is continued until the candidate list L becomes empty.

The (approximate) boundary of S can be found in list B. The other output list, T, contains some "garbage" patches which may be the subject of further geometric calculations or may simply be discarded.

The crucial point is how to classify the patches with respect to the solid. The cited authors propose a computationally not too expensive approximate solution to this problem, which they call the method of *inner sets and outer sets*; the ISOS method.
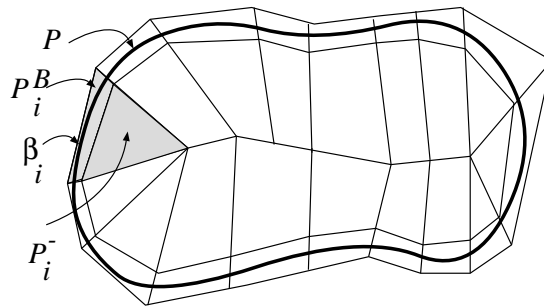
Figure 4.6: Inner and outer segments

Each primitive object P is approximated by two polyhedra: an *inner* polyhedron P^- and an *outer* polyhedron P^+:

$$P^- \subseteq P \subseteq P^+. \tag{4.14}$$

Both polyhedra are constructed from polyhedral segments. The segments of P^- and P^+, however, are not independent of each other, as illustrated in figure 4.6. The outer polyhedron P^+ consists of the *outer segments*, say $P_1^+, \ldots, P_n^+$. An outer segment P_i^+ is the union of two subsegments: the *inner segment P_i^-*, which is totally contained by the primitive, and the *boundary segment P_i^B*, which contains a boundary patch, say β_i (a part of the boundary of the primitive). The thinner the boundary segments the better the approximation of the primitive boundary by the union of the boundary segments. A coarse decomposition of each primitive is created in a preliminary phase according to point 1 of the above outlined strategy.

Set membership classification of a boundary patch β with respect to the compound solid S (point 2) is approximated by means of the inner, outer and boundary segments corresponding to the primitives. According to the **divide-and-conquer** approach, two different cases can be distinguished: one in which S is primitive and the second is in which S is compound.

Case 1: Classification of β with respect to a primitive P
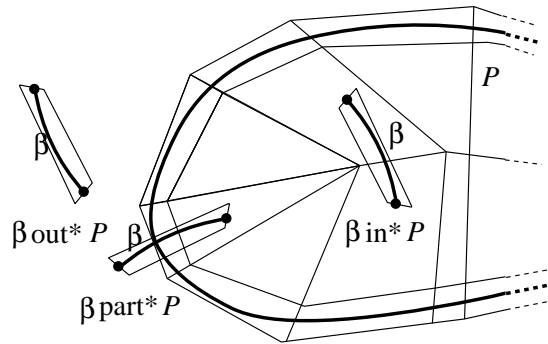
Figure 4.7: Relations between a boundary segment and a primitive

The following examinations must be made on the boundary segment P^B containing the boundary patch β with respect to P in this order (it is assumed that β is not a boundary patch of P because this case can be detected straightforwardly):

1. Test whether P^B intersects any of the outer segments $P_1^+, \ldots, P_n^+$ corresponding to P. If the answer is negative, then $P^B \, \text{out}^* \, P$ holds,

that is $\beta\,\text{out}^*\,P$ (see figure 4.7). Otherwise go to examination 2 (β is either totally or partially contained by P).

2. Test whether P^B intersects any of the boundary segments $P_1^B,\ldots,P_n^B$ corresponding to P. If the answer is negative, then $P^B\,\text{in}^*\,P$ holds, that is $\beta\,\text{in}^*\,P$ (see figure 4.7). Otherwise go to examination 3.

3. In this case, due to the polyhedral approximation, nothing more can be stated about β, that is, either one or none of the relations $\beta\,\text{in}^*\,P$, $\beta\,\text{out}^*\,P$ and (accidentally) $\beta\,\text{on}^*\,P$ could hold (figure 4.7 shows a situation where none of them holds). β is classified as *partial* in this case. This is expressed by the notation $\beta\,\text{part}^*\,P$ (according to point 4 of the generate-and-test strategy outlined previously, β will then be subdivided).

Classification results with respect to two primitives connected by a set operation can then be *combined*.

Case 2: Classification of β with respect to $S = A \circ^* B$
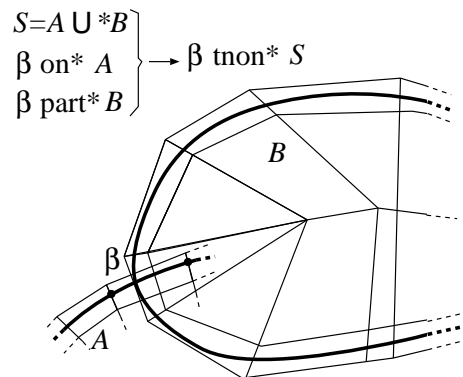
Figure 4.8: A boundary segment classified as a tentative boundary

After computing the classification of β with respect to A and B, the two results can be combined according to the following tables (new notations are defined after):

$S = A \cup^* B$	$\beta\,\text{in}^* B$	$\beta\,\text{out}^* B$	$\beta\,\text{on}^* B$	$\beta\,\text{part}^* B$	$\beta\,\text{tnon}^* B$
$\beta\,\text{in}^* A$	$\beta\,\text{in}^* S$	$\beta\,\text{in}^* S$	$\beta\,\text{in}^* S$	$\beta\,\text{in}^* S$	$\beta\,\text{in}^* S$
$\beta\,\text{out}^* A$	$\beta\,\text{in}^* S$	$\beta\,\text{out}^* S$	$\beta\,\text{on}^* S$	$\beta\,\text{part}^* S$	$\beta\,\text{tnon}^* S$
$\beta\,\text{on}^* A$	$\beta\,\text{in}^* S$	$\beta\,\text{on}^* S$	$*$	$\beta\,\text{tnon}^* S$	$*$
$\beta\,\text{part}^* A$	$\beta\,\text{in}^* S$	$\beta\,\text{part}^* S$	$\beta\,\text{tnon}^* S$	$\beta\,\text{part}^* S$	$\beta\,\text{tnon}^* S$
$\beta\,\text{tnon}^* A$	$\beta\,\text{in}^* S$	$\beta\,\text{tnon}^* S$	$*$	$\beta\,\text{tnon}^* S$	$*$

$S = A \cap^* B$	$\beta\,\text{in}^* B$	$\beta\,\text{out}^* B$	$\beta\,\text{on}^* B$	$\beta\,\text{part}^* B$	$\beta\,\text{tnon}^* B$
$\beta\,\text{in}^* A$	$\beta\,\text{in}^* S$	$\beta\,\text{out}^* S$	$\beta\,\text{on}^* S$	$\beta\,\text{part}^* S$	$\beta\,\text{tnon}^* S$
$\beta\,\text{out}^* A$	$\beta\,\text{out}^* S$	$\beta\,\text{out}^* S$	$\beta\,\text{out}^* S$	$\beta\,\text{out}^* S$	$\beta\,\text{out}^* S$
$\beta\,\text{on}^* A$	$\beta\,\text{on}^* S$	$\beta\,\text{out}^* S$	$*$	$\beta\,\text{tnon}^* S$	$*$
$\beta\,\text{part}^* A$	$\beta\,\text{part}^* S$	$\beta\,\text{out}^* S$	$\beta\,\text{tnon}^* S$	$\beta\,\text{part}^* S$	$\beta\,\text{tnon}^* S$
$\beta\,\text{tnon}^* A$	$\beta\,\text{tnon}^* S$	$\beta\,\text{out}^* S$	$*$	$\beta\,\text{tnon}^* S$	$*$

$S = A \setminus^* B$	$\beta\,\text{in}^* B$	$\beta\,\text{out}^* B$	$\beta\,\text{on}^* B$	$\beta\,\text{part}^* B$	$\beta\,\text{tnon}^* B$
$\beta\,\text{in}^* A$	$\beta\,\text{out}^* S$	$\beta\,\text{in}^* S$	$\beta\,\text{on}^* S$	$\beta\,\text{part}^* S$	$\beta\,\text{tnon}^* S$
$\beta\,\text{out}^* A$	$\beta\,\text{out}^* S$	$\beta\,\text{out}^* S$	$\beta\,\text{out}^* S$	$\beta\,\text{out}^* S$	$\beta\,\text{out}^* S$
$\beta\,\text{on}^* A$	$\beta\,\text{out}^* S$	$\beta\,\text{on}^* S$	$*$	$\beta\,\text{tnon}^* S$	$*$
$\beta\,\text{part}^* A$	$\beta\,\text{out}^* S$	$\beta\,\text{part}^* S$	$\beta\,\text{tnon}^* S$	$\beta\,\text{part}^* S$	$\beta\,\text{tnon}^* S$
$\beta\,\text{tnon}^* A$	$\beta\,\text{out}^* S$	$\beta\,\text{tnon}^* S$	$*$	$\beta\,\text{tnon}^* S$	$*$

Two new notations are used here in addition to those already introduced. The notation $\beta\,\text{tnon}^* S$ is used to express that β is a *tentative boundary* patch (see figure 4.8). The use of this result in the classification scheme always happens at a stage where one of the classification results to be combined is "on*" and the other is "part*", in which case the relation of β with respect to S cannot be ascertained. The patch can then be the subject of subdivision and some of the subpatches *may* come out as boundary patches. The other notation, the asterisk $(*)$, denotes that the situation can occur in case of degeneracies, and that special care should then be taken in order to resolve degeneracies so that regularity of the set operations is not violated (this requires further geometric calculations).

Chapter 5

TRANSFORMATIONS, CLIPPING AND PROJECTION

5.1   Geometric transformations

Three-dimensional graphics aims at producing an image of 3D objects. This means that the geometrical representation of the image is generated from the geometrical data of the objects. This change of geometrical description is called the **geometric transformation**. In computers the world is represented by numbers; thus geometrical properties and transformations must also be given by numbers in computer graphics. Cartesian coordinates provide this algebraic establishment for the Euclidean geometry, which define a 3D point by three component distances along three, non-coplanar axes from the origin of the coordinate system.

The selection of the origin and the axes of this coordinate system may have a significant effect on the complexity of the definition and various calculations. As mentioned earlier, the world coordinate system is usually not suitable for the definition of all objects, because here we are not only concerned with the geometry of the objects, but also with their relative position and orientation. A brick, for example, can be simplistically defined in a coordinate system having axes parallel to its edges, but the description of the box is quite complicated if arbitrary orientation is required. This consid-

eration necessitated the application of local coordinate systems. Viewing and visibility calculations, on the other hand, have special requirements from a coordinate system where the objects are represented, to facilitate simple operations. This means that the definition and the photographing of the objects may involve several different coordinate systems suitable for the different specific operations. The transportation of objects from one coordinate system to another also requires geometric transformations.

Working in several coordinate systems can simplify the various phases of modeling and image synthesis, but it requires additional transformation steps. Thus, this approach is advantageous only if the computation needed for geometric transformations is less than the decrease of the computation of the various steps due to the specifically selected coordinate systems. Representations invariant of the transformations are the primary candidates for methods working in several coordinate systems, since they can easily be transformed by transforming the control or definition points. Polygon mesh models, Bezier and B-spline surfaces are invariant for linear transformation, since their transformation will also be polygon meshes, Bezier or B-spline surfaces, and the vertices or the control points of the transformed surface will be those coming from the transformation of the original vertices and control points.

Other representations, sustaining non-planar geometry, and containing, for example, spheres, are not easily transformable, thus they require all the calculations to be done in a single coordinate system.

Since computer graphics generates 2D images of 3D objects, some kind of projection is always involved in image synthesis. Central projection, however, creates problems (singularities) in Euclidean geometry, it is thus worthwhile considering another geometry, namely the **projective geometry**, to be used for some phases of image generation. Projective geometry is a classical branch of mathematics which cannot be discussed here in detail. A short introduction, however, is given to highlight those features that are widely used in computer graphics. Beyond this elementary introduction, the interested reader is referred to [Her91] [Cox74].

Projective geometry can be approached from the analysis of central projection as shown in figure 5.1.

For those points to which the projectors are parallel with the image plane no projected image can be defined in Euclidean geometry. Intuitively speaking these image points would be at "infinity" which is not part of the Eu-
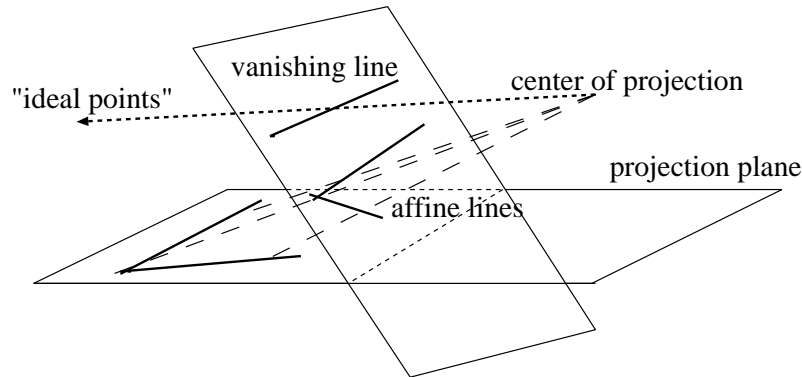
Figure 5.1: Central projection of objects on a plane

clidean space. Projective geometry fills these holes by extending the Euclidean space by new points, called **ideal points**, that can serve as the image of points causing singularities in Euclidean space. These ideal points can be regarded as "intersections" of parallel lines and planes, which are at "infinity". These ideal points form a plane of the projective space, which is called the **ideal plane**.

Since there is a one-to-one correspondence between the points of Euclidean space and the coordinate triples of a Cartesian coordinate system, the new elements obviously cannot be represented in this coordinate system, but a new algebraic establishment is needed for projective geometry. This establishment is based on **homogeneous coordinates**.

For example, by the method of homogeneous coordinates a point of space can be specified as the center of gravity of the structure containing mass X_h at reference point p_1, mass Y_h at point p_2, mass Z_h at point p_3 and mass w at point p_4. Weights are not required to be positive, thus the center of gravity can really be any point of the space if the four reference points are not co-planar. Alternatively, if the total mass, that is $h = X_h + Y_h + Z_h + w$, is not zero and the reference points are in Euclidean space, then the center of gravity will also be in the Euclidean space.

Let us call the quadruple (X_h, Y_h, Z_h, h), where $h = X_h + Y_h + Z_h + w$, the homogeneous coordinates of the center of gravity.

Note that if all weights are multiplied by the same (non-zero) factor, the center of gravity, that is the point defined by the homogeneous coordi-

nates, does not change. Thus a point (X_h, Y_h, Z_h, h) is equivalent to points $(\lambda X_h, \lambda Y_h, \lambda Z_h, \lambda h)$, where λ is a non-zero number.

The center of gravity analogy used to illustrate the homogeneous coordinates is not really important from a mathematical point of view. What should be remembered, however, is that a 3D point represented by homogeneous coordinates is a four-vector of real numbers and all scalar multiples of these vectors are equivalent.

Points of the projective space, that is the points of the Euclidean space (also called **affine points**) plus the ideal points, can be represented by homogeneous coordinates. First the representation of affine points which can establish a correspondence between the Cartesian and the homogeneous coordinate systems is discussed. Let us define the four reference points of the homogeneous coordinate system in points [1,0,0], [0,1,0], [0,0,1] and in [0,0,0] respectively. If $h = X_h + Y_h + Z_h + w$ is not zero, then the center of gravity in Cartesian coordinate system defined by axes $\mathbf{i}, \mathbf{j}, \mathbf{k}$ is:

$$r(X_h, Y_h, Z_h, h) = \frac{1}{h}(X_h \cdot [1,0,0] + Y_h \cdot [0,1,0] + Z_h \cdot [0,0,1] + w \cdot [0,0,0]) =$$

$$\frac{X_h}{h} \cdot \mathbf{i} + \frac{Y_h}{h} \cdot \mathbf{j} + \frac{Z_h}{h} \cdot \mathbf{k}. \tag{5.1}$$

Thus with the above selection of reference points the correspondence between the homogeneous coordinates (X_h, Y_h, Z_h, h) and Cartesian coordinates (x, y, z) of affine points $(h \neq 0)$ is:

$$x = \frac{X_h}{h}, \qquad y = \frac{Y_h}{h}, \qquad z = \frac{Z_h}{h}. \tag{5.2}$$

Homogeneous coordinates can also be used to characterize planes. In the Cartesian system a plane is defined by the following equation:

$$a \cdot x + b \cdot y + c \cdot z + d = 0 \tag{5.3}$$

Applying the correspondence between the homogeneous and Cartesian coordinates, we get:

$$a \cdot X_h + b \cdot Y_h + c \cdot Z_h + d \cdot h = 0 \tag{5.4}$$

Note that the set of points that satisfy this plane equation remains the same if this equation is multiplied by a scalar factor. Thus a quadruple $[a, b, c, d]$

of homogeneous coordinates can represent not only single points but planes as well. In fact all theorems valid for points can be formulated for planes as well in 3D projective space. This symmetry is often referred to as the **duality principle**. The intersection of two planes (which is a line) can be calculated as the solution of the linear system of equations. Suppose that we have two parallel planes given by quadruples $[a, b, c, d]$ and $[a, b, c, d']$ ($d \neq d'$) and let us calculate their intersection. Formally all points satisfy the resulting equations for which

$$a \cdot X_h + b \cdot Y_h + c \cdot Z_h = 0 \quad \text{and} \quad h = 0 \tag{5.5}$$

In Euclidean geometry parallel planes do not have intersection, thus the points calculated in this way cannot be in Euclidean space, but form a subset of the ideal points of the projective space. This means that ideal points correspond to those homogeneous quadruples where $h = 0$. As mentioned, these ideal points represent the infinity, but they make a clear distinction between the "infinities" in different directions that are represented by the first three coordinates of the homogeneous form.

Returning to the equation of a projective plane or considering the equation of a projective line, we can realize that ideal points may also satisfy these equations. Therefore, projective planes and lines are a little bit more than their Euclidean counterparts. In addition to all Euclidean points, they also include some ideal points. This may cause problems when we want to return to Euclidean space because these ideal points have no counterparts.

Homogeneous coordinates can be visualized by regarding them as Cartesian coordinates of a higher dimensional space (note that 3D points are defined by 4 homogeneous coordinates). This procedure is called the embedding of the 3D projective space into the 4D Euclidean space or the **straight model** [Her91] (figure 5.2). Since it is impossible to create 4D drawings, this visualization uses a trick of reducing the dimensionality and displays the 4D space as a 3D one, the real 3D subspace as a 2D plane and relies on the reader's imagination to interpret the resulting image.

A homogeneous point is represented by a set of equivalent quadruples

$$\{(\lambda X_h, \lambda Y_h, \lambda Z_h, \lambda h) \mid \lambda \neq 0\},$$

thus a point is described as a 4D line crossing the origin, [0,0,0,0], in the straight model. Ideal points are in the $h = 0$ plane and affine points are represented by those lines that are not parallel to the $h = 0$ plane.
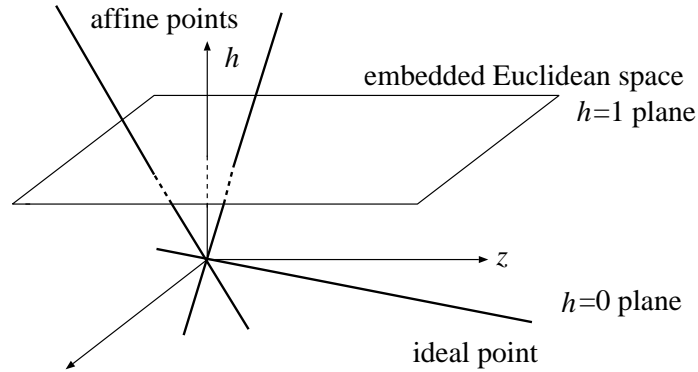
Figure 5.2: Embedding of projective space into a higher dimensional Euclidean space

Since points are represented by a set of quadruples that are equivalent in homogeneous terms, a point may be represented by any of them. Still, it is worth selecting a single representative from this set to identify points unambiguously. For affine points, this representative quadruple is found by making the fourth (h) coordinate equal to 1, which has a nice property that the first three homogeneous coordinates are equal to the Cartesian coordinates of the same point taking equation 5.2 into account, that is:

$$(\frac{X_h}{h}, \frac{Y_h}{h}, \frac{Z_h}{h}, 1) = (x, y, z, 1). \tag{5.6}$$

In the straight model thus the representatives of affine points correspond to the $h = 1$ hyperplane (a 3D set of the 4D space), where they can be identified by Cartesian coordinates. This can be interpreted as the 3D Euclidean space and for affine points the homogeneous to Cartesian conversion of coordinates can be accomplished by projecting the 4D point onto the $h = 1$ hyperplane using the origin as the center of projection. This projection means the division of the first three coordinates by the fourth and is usually called **homogeneous division**.

Using the algebraic establishment of Euclidean and projective geometries, that is the system of Cartesian and homogeneous coordinates, geometric transformations can be regarded as functions that map tuples of coordinates onto tuples of coordinates. In computer graphics linear functions are

preferred that can conveniently be expressed as a vector-matrix multiplication and a vector addition. In Euclidean geometry this linear function has the following general form:

$$[x', y', z'] = [x, y, z] \cdot \mathbf{A_{3\times3}} + [p_x, p_y, p_z]. \tag{5.7}$$

Linear transformations of this kind map affine points onto affine points, therefore they are also **affine transformations**.

When using homogeneous representation, however, it must be taken into account that equivalent quadruples differing only by a scalar multiplication must be transformed to equivalent quadruples, thus no additive constant is allowed:

$$[X'_h, Y'_h, Z'_h, h'] = [X_h, Y_h, Z_h, h] \cdot \mathbf{T_{4\times4}}. \tag{5.8}$$

Matrix $\mathbf{T_{4\times4}}$ defines the transformation uniquely in homogeneous sense; that is, matrices differing in a multiplicative factor are equivalent.

Note that in equations 5.7 and 5.8 row vectors are used to identify points unlike the usual mathematical notation. The preference for row vectors in computer graphics has partly historical reasons, partly stems from the property that in this way the concatenation of transformations corresponds to matrix multiplication in "normal", that is left to right, order. For column vectors, it would be the reverse order. Using the straight model, equation 5.7 can be reformulated for homogeneous coordinates:

$$[x', y', z', 1] = [x, y, z, 1] \cdot \begin{bmatrix} & & & 0 \\ \mathbf{A}_{3\times3} & & 0 \\ & & & 0 \\ \mathbf{p^T} & & & 1 \end{bmatrix}. \tag{5.9}$$

Note that the 3×3 matrix $\mathbf{A}$ is accommodated in $\mathbf{T}$ as its upper left minor matrix, while $\mathbf{p}$ is placed in the last row and the fourth column vector of $\mathbf{T}$ is set to constant [0,0,0,1]. This means that the linear transformations of Euclidean space form a subset of homogeneous linear transformations. This is a real subset since, as we shall see, by setting the fourth column to a vector different from [0,0,0,1] the resulting transformation does not have an affine equivalent, that is, it is not linear in the Euclidean space.

Using the algebraic treatment of homogeneous (linear) transformations, which identifies them by a 4×4 matrix multiplication, we can define the concatenation of transformations as the product of transformation matrices

and the inverse of a homogeneous transformation as the inverse of its transformation matrix if it exists, i.e. its determinant is not zero. Taking into account the properties of matrix operations we can see that the concatenation of homogeneous transformations is also a homogeneous transformation and the inverse of a homogeneous transformation is also a homogeneous transformation if the transformation matrix is invertible. Since matrix multiplication is an associative operation, consecutive transformations can always be replaced by a single transformation by computing the product of the matrices of different transformation steps. Thus, any number of linear transformations can be expressed by a single 4×4 matrix multiplication. The transformation of a single point of the projective space requires 16 multiplications and 12 additions. If the point must be mapped back to the Cartesian coordinate system, then 3 divisions by the fourth homogeneous coordinate may be necessary in addition to the matrix multiplication. Since linear transformations of Euclidean space have a $[0, 0, 0, 1]$ fourth column in the transformation matrix, which is preserved by multiplications with matrices of the same property, any linear transformation can be calculated by 9 multiplications and 9 additions.

According to the theory of projective geometry, transformations defined by 4×4 matrix multiplication map points onto points, lines onto lines, planes onto planes and intersection points onto intersection points, and therefore are called **collinearities** [Her91]. The reverse of this statement is also true; each collinearity corresponds to a homogeneous transformation matrix. Instead of proving this statement in projective space, a special case that has importance in computer graphics is investigated in detail. In computer graphics the geometry is given in 3D Euclidean space and having applied some homogeneous transformation the results are also required in Euclidean space. From this point of view, the homogeneous transformation of a 3D point involves:

1. A 4×4 matrix multiplication of the coordinates extended by a fourth coordinate of value 1.

2. A homogeneous division of all coordinates in the result by the fourth coordinate if it is different from 1, meaning that the transformation forced the point out of 3D space.

It is important to note that a clear distinction must be made between the

central or parallel projection defined earlier which maps 3D points onto 2D points on a plane and projective transformations which map projective space onto projective space. Now let us start the discussion of the homogeneous transformation of a special set of geometric primitives. A Euclidean line can be defined by the following equation:

$$\vec{r}(t) = \vec{r}_0 + \vec{v} \cdot t, \quad \text{where } t \text{ is a real parameter.} \tag{5.10}$$

Assuming that vectors $\vec{v}_1$ and $\vec{v}_2$ are not parallel, a Euclidean plane, on the other hand, can be defined as follows:

$$\vec{r}(t_1, t_2) = \vec{r}_0 + \vec{v}_1 \cdot t_1 + \vec{v}_2 \cdot t_2, \quad \text{where } t_1, t_2 \text{ are real parameters.} \tag{5.11}$$

Generally, lines and planes are special cases of a wider range of geometric structures called linear sets. By definition, a **linear set** is defined by a position vector $\vec{r}_0$ and some axes $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$ by the following equation:

$$\vec{r}(t_1, \ldots, t_n) = \vec{r}_0 + \sum_{i=1}^{n} t_i \cdot \vec{v}_i. \tag{5.12}$$

First of all, the above definition is converted to a different one that uses homogeneous-like coordinates. Let us define the so-called spanning vectors $\vec{p}_0, \ldots, \vec{p}_n$ of the linear set as:

$$
\begin{aligned}
\vec{p}_0 &= \vec{r}_0, \\
\vec{p}_1 &= \vec{r}_0 + \vec{v}_1, \\
&\vdots \\
\vec{p}_n &= \vec{r}_0 + \vec{v}_n.
\end{aligned}
\tag{5.13}
$$

The equation of the linear set is then:

$$\vec{r}(t_1, \ldots, t_n) = (1 - t_1 - \ldots - t_n) \cdot \vec{p}_0 + \sum_{i=1}^{n} t_i \cdot \vec{p}_i. \tag{5.14}$$

Introducing the new coordinates as

$$\alpha_0 = 1 - t_1 - \ldots - t_n, \quad \alpha_1 = t_1, \quad \alpha_2 = t_2, \quad \ldots, \quad \alpha_n = t_n, \tag{5.15}$$

the linear set can be written in the following form:

$$S = \{\vec{p} \mid \vec{p} = \sum_{i=0}^{n} \alpha_i \cdot \vec{p}_i \ \wedge \ \sum_{i=0}^{n} \alpha_i = 1\}. \tag{5.16}$$

The weights (α_i) are also called the **baricentric coordinates** of the point $\vec{p}$ with respect to $\vec{p}_0$, $\vec{p}_1$,...,$\vec{p}_n$. This name reflects the interpretation that $\vec{p}$ would be the center of gravity of a structure of weights $(\alpha_0, \alpha_1, \ldots, \alpha_n)$ at points $\vec{p}_0, \vec{p}_1, \ldots, \vec{p}_n$.

The homogeneous transformation of such a point $\vec{p}$ is:

$$[\vec{p}, 1] \cdot \mathbf{T} = [\sum_{i=0}^{n} \alpha_i \cdot \vec{p}_i, 1] \cdot \mathbf{T} = [\sum_{i=0}^{n} \alpha_i \cdot \vec{p}_i, \sum_{i=0}^{n} \alpha_i] \cdot \mathbf{T} =$$

$$(\sum_{i=0}^{n} \alpha_i \cdot [\vec{p}_i, 1]) \cdot \mathbf{T} = \sum_{i=0}^{n} \alpha_i \cdot ([\vec{p}_i, 1] \cdot \mathbf{T}) \qquad (5.17)$$

since $\sum_{i=0}^{n} \alpha_i = 1$. Denoting $[\vec{p}_i, 1] \cdot \mathbf{T}$ by $[\vec{P}_i, h_i]$ we get:

$$[\vec{p}, 1] \cdot \mathbf{T} = \sum_{i=0}^{n} \alpha_i \cdot [\vec{P}_i, h_i] = [\sum_{i=0}^{n} \alpha_i \cdot \vec{P}_i, \sum_{i=0}^{n} \alpha_i \cdot h_i]. \qquad (5.18)$$

If the resulting fourth coordinate $\sum_{i=0}^{n} \alpha_i \cdot h_i$ is zero, then the point $\vec{p}$ is mapped onto an ideal point, therefore it cannot be converted back to Euclidean space. These ideal points must be eliminated before the homogeneous division (see section 5.5 on clipping).

After homogeneous division we are left with:

$$[\sum_{i=0}^{n} \frac{\alpha_i \cdot h_i}{\sum_{j=1}^{n} \alpha_j \cdot h_j} \cdot \frac{\vec{P}_i}{h_i}, 1] = [\sum_{i=0}^{n} \alpha_i^* \cdot \vec{p}_i^{\,*}, 1] \qquad (5.19)$$

where $\vec{p}_i^{\,*}$ is the homogeneous transformation of $\vec{p}_i$. The derivation of α_i^* guarantees that $\sum_{i=0}^{n} \alpha_i^* = 1$. Thus, the transformation of the linear set is also linear. Examining the expression of the weights (α_i^*), we can conclude that generally $\alpha_i \neq \alpha_i^*$ meaning the homogeneous transformation may destroy equal spacing. In other words the **division ratio** is not projective invariant. In the special case when the transformation is affine, coordinates h_i will be 1, thus $\alpha_i = \alpha_i^*$, which means that equal spacing (or division ratio) is affine invariant.

A special type of linear set is the **convex hull**. The convex hull is defined by equation 5.16 with the provision that the baricentric coordinates must be non-negative.

To avoid the problems of mapping onto ideal points, let us assume the spanning vectors to be mapped onto the same side of the $h = 0$ hyperplane, meaning that the h_i-s must have the same sign. This, with $\alpha_i \geq 0$, guarantees that no points are mapped onto ideal points and

$$\alpha_i^* = \sum_{i=0}^{n} \frac{\alpha_i \cdot h_i}{\sum_{i=0}^{n} \alpha_i \cdot h_i} \geq 0 \qquad (5.20)$$

Thus, baricentric coordinates of the image will also be non-negative, that is, convex hulls are also mapped onto convex hulls by homogeneous transformations if their transformed image does not contain ideal points. An arbitrary planar polygon can be broken down into triangles that are convex hulls of three spanning vectors. The transformation of this polygon will be the composition of the transformed triangles. This means that a planar polygon will also be preserved by homogeneous transformations if its image does not intersect with the $h = 0$ plane.

As mentioned earlier, in computer graphics the objects are defined in Euclidean space by Cartesian coordinates and the image is required in a 2D pixel space that is also Euclidean with its coordinates which correspond to the physical pixels of the frame buffer. Projective geometry may be needed only for specific stages of the transformation from modeling to pixel space. Since projective space can be regarded as an extension of the Euclidean space, the theory of transformations could be discussed generally only in projective space. For pedagogical reasons, however, we will use the more complicated homogeneous representations if they are really necessary for computer graphics algorithms, and deal with the Cartesian coordinates in simpler cases. This combined view of Euclidean and projective geometries may be questionable from a purely mathematical point of view, but it is accepted by the computer graphics community because of its clarity and its elimination of unnecessary abstractions.

We shall consider the transformation of points in this section, which will lead on to the transformation of planar polygons as well.

5.1.1   Elementary transformations

Translation

Translation is a very simple transformation that adds the translation vector $\vec{p}$ to the position vector $\vec{r}$ of the point to be transformed:

$$\vec{r}\,' = \vec{r} + \vec{p}. \tag{5.21}$$

Scaling along the coordinate axes

Scaling modifies the distances and the size of the object independently along the three coordinate axes. If a point originally has $[x, y, z]$ coordinates, for example, after scaling the respective coordinates are:

$$x' = S_x \cdot x, \qquad y' = S_y \cdot y, \qquad z' = S_z \cdot z. \tag{5.22}$$

This transformation can also be expressed by a matrix multiplication:

$$\vec{r}\,' = \vec{r} \cdot \left[\begin{array}{ccc} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{array} \right]. \tag{5.23}$$

Rotation around the coordinate axes

Rotating around the z axis by an angle ϕ, the x and y coordinates of a point are transformed according to figure 5.3, leaving coordinate z unaffected.
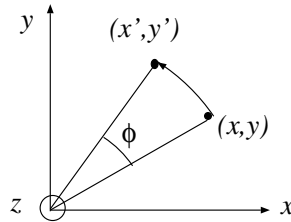
Figure 5.3: Rotation around the z axis

By geometric considerations, the new x, y coordinates can be expressed as:

$$x' = x \cdot \cos \phi - y \cdot \sin \phi, \qquad y' = x \cdot \sin \phi + y \cdot \cos \phi. \tag{5.24}$$

Rotations around the y and x axes have similar form, just the roles of x, y and z must be exchanged. These formulae can also be expressed in matrix form:

$$\vec{r}\,'(x, \phi) = \vec{r} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}$$

$$\vec{r}\,'(y, \phi) = \vec{r} \cdot \begin{bmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{bmatrix} \tag{5.25}$$

$$\vec{r}\,'(z, \phi) = \vec{r} \cdot \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

These rotations can be used to express any orientation [Lan91]. Suppose that K and K''' are two Cartesian coordinate systems sharing a common origin but having different orientations. In order to determine three special rotations around the coordinate axes which transform K into K''', let us define a new Cartesian system K' such that its z' axis is coincident with z and its y' axis is on the intersection line of planes $[x, y]$ and $[x''', y''']$. To transform axis y onto axis y' a rotation is needed around z by angle α. Then a new rotation around y' by angle β has to be applied that transforms x' into x''' resulting in a coordinate system K''. Finally the coordinate system K'' is rotated around axis $x'' = x'''$ by an angle γ to transform y'' into y'''.

The three angles, defining the final orientation, are called **roll, pitch and yaw angles**. If the roll, pitch and yaw angles are α, β and γ respectively, the transformation to the new orientation is:

$$\vec{r}\,' = \vec{r} \cdot \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & \sin\gamma \\ 0 & -\sin\gamma & \cos\gamma \end{bmatrix}.$$
$$\tag{5.26}$$

Rotation around an arbitrary axis

Let us examine a linear transformation that corresponds to a rotation by angle ϕ around an arbitrary unit axis $\vec{t}$ going through the origin. The original and the transformed points are denoted by vectors $\vec{u}$ and $\vec{v}$ respectively.

Let us decompose vectors $\vec{u}$ and $\vec{v}$ into perpendicular $(\vec{u}_\perp, \vec{v}_\perp)$ and parallel $(\vec{u}_\parallel, \vec{v}_\parallel)$ components with respect to $\vec{t}$. By geometrical considerations we can write:

$$
\begin{aligned}
\vec{u}_\parallel &= \vec{t}(\vec{t} \cdot \vec{u}) \\
\vec{u}_\perp &= \vec{u} - \vec{u}_\parallel = \vec{u} - \vec{t}(\vec{t} \cdot \vec{u})
\end{aligned}
\tag{5.27}
$$

Since the rotation does not affect the parallel component, $\vec{v}_\parallel = \vec{u}_\parallel$.
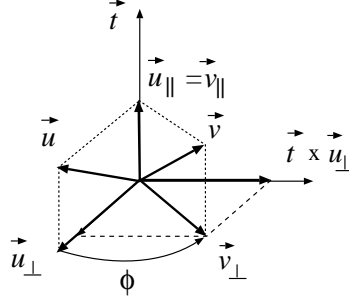
Figure 5.4: Rotating around $\vec{t}$ by angle ϕ

Since vectors $\vec{u}_\perp, \vec{v}_\perp$ and $\vec{t} \times \vec{u}_\perp = \vec{t} \times \vec{u}$ are in the plane perpendicular to $\vec{t}$, and $\vec{u}_\perp$ and $\vec{t} \times \vec{u}_\perp$ are perpendicular vectors (figure 5.4), $\vec{v}_\perp$ can be expressed as:

$$
\vec{v}_\perp = \vec{u}_\perp \cdot \cos\phi + \vec{t} \times \vec{u}_\perp \cdot \sin\phi.
\tag{5.28}
$$

Vector $\vec{v}$, that is the rotation of $\vec{u}$, can then be expressed as follows:

$$
\vec{v} = \vec{v}_\parallel + \vec{v}_\perp = \vec{u} \cdot \cos\phi + \vec{t} \times \vec{u} \cdot \sin\phi + \vec{t}(\vec{t} \cdot \vec{u})(1 - \cos\phi).
\tag{5.29}
$$

This equation, also called the **Rodrigues formula**, can also be expressed in matrix form. Denoting $\cos\phi$ and $\sin\phi$ by C_ϕ and S_ϕ respectively and assuming $\vec{t}$ to be a unit vector, we get:

$$
\vec{v} = \vec{u} \cdot \begin{bmatrix}
C_\phi(1 - t_x^2) + t_x^2 & t_x t_y(1 - C_\phi) + S_\phi t_z & t_x t_z(1 - C_\phi) - S_\phi t_y \\
t_y t_x(1 - C_\phi) - S_\phi t_z & C_\phi(1 - t_y^2) + t_y^2 & t_x t_z(1 - C_\phi) + S_\phi t_x \\
t_z t_x(1 - C_\phi) + S_\phi t_y & t_z t_y(1 - C_\phi) - S_\phi t_x & C_\phi(1 - t_z^2) + t_z^2
\end{bmatrix}.
\tag{5.30}
$$

It is important to note that any orientation can also be expressed as a rotation around an appropriate axis. Thus, there is a correspondence between roll-pitch-yaw angles and the axis and angle of final rotation, which can be given by making the two transformation matrices defined in equations 5.26 and 5.30 equal and solving the equation for unknown parameters.

Shearing

Suppose a shearing stress acts on a block fixed on the xy face of figure 5.5, deforming the block to a parallepiped. The transformation representing the distortion of the block leaves the z coordinate unaffected, and modifies the x and y coordinates proportionally to the z coordinate.
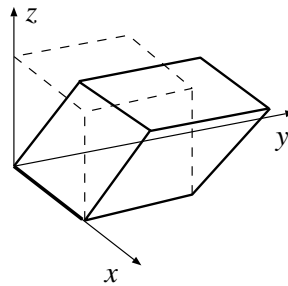
Figure 5.5: Shearing of a block

In matrix form the shearing transformation is:

$$\vec{r}\,' = \vec{r} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}. \tag{5.31}$$

5.2 Transformation to change the coordinate system

Objects defined in one coordinate system are often needed in another coordinate system. When we decide to work in several coordinate systems and to make every calculation in the coordinate system in which it is the

simplest, the coordinate system must be changed for each different phase of the calculation.

Suppose unit coordinate vectors $\vec{u}$, $\vec{v}$ and $\vec{w}$ and the origin $\vec{o}$ of the new coordinate system are defined in the original x, y, z coordinate system:

$$\vec{u} = [u_x, u_y, u_z], \quad \vec{v} = [v_x, v_y, v_z], \quad \vec{w} = [w_x, w_y, w_z], \quad \vec{o} = [o_x, o_y, o_z]. \quad (5.32)$$

Let a point $\vec{p}$ have x, y, z and α, β, γ coordinates in the x, y, z and in the u, v, w coordinate systems respectively. Since the coordinate vectors $\vec{u}, \vec{v}, \vec{w}$ as well as their origin, $\vec{o}$, are known in the x, y, z coordinate system, $\vec{p}$ can be expressed in two different forms:

$$\vec{p} = \alpha \cdot \vec{u} + \beta \cdot \vec{v} + \gamma \cdot \vec{w} + \vec{o} = [x, y, z]. \quad (5.33)$$

This equation can also be written in homogeneous matrix form, having introduced the matrix formed by the coordinates of the vectors defining the u, v, w coordinate system:

$$\mathbf{T_c} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ o_x & o_y & o_z & 1 \end{bmatrix}, \quad (5.34)$$

$$[x, y, z, 1] = [\alpha, \beta, \gamma, 1] \cdot \mathbf{T_c}. \quad (5.35)$$

Since $\mathbf{T_c}$ is always invertible, the coordinates of a point of the x, y, z coordinate system can be expressed in the u, v, w coordinate system as well:

$$[\alpha, \beta, \gamma, 1] = [x, y, z, 1] \cdot \mathbf{T_c}^{-1}. \quad (5.36)$$

Note that the inversion of matrix $\mathbf{T_c}$ can be calculated quite effectively since its upper-left minor matrix is orthonormal, that is, its inverse is given by mirroring the matrix elements onto the diagonal of the matrix, thus:

$$\mathbf{T_c}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -o_x & -o_y & -o_z & 1 \end{bmatrix} \cdot \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (5.37)$$

5.3   Definition of the camera

Having defined transformation matrices we can now look at their use in image generation, but first some basic definitions.

In 3D image generation, a **window** rectangle is placed into the 3D space of the virtual world with arbitrary orientation, a **camera** or **eye** is put behind the window, and a photo is taken by projecting the model onto the window plane, supposing the camera to be the center of projection, and ignoring the parts mapped outside the window rectangle or those which are not in the specified region in front of the camera. The data, which define how the virtual world is looked at, are called **camera parameters**, and include:
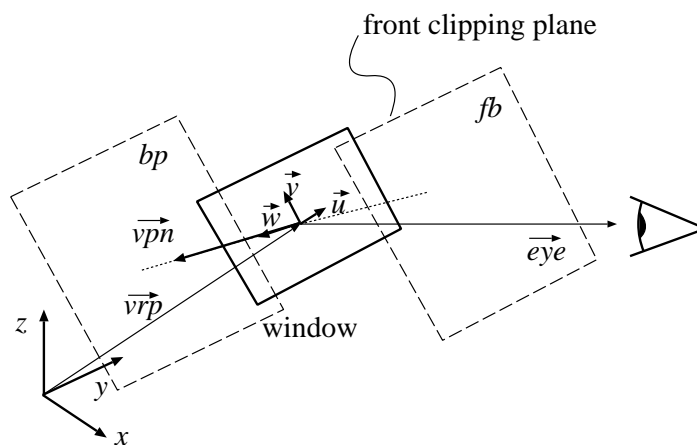
Figure 5.6: Definition of the camera

- **Position and orientation of the window**. The center of the window, called the **view reference point**, is defined as a point, or a vector $v\vec{r}p$, in the world coordinate system. The orientation is defined by a u, v, w orthogonal coordinate system, which is also called the **window coordinate system**, centered at the view reference point, with $\vec{u}$ and $\vec{v}$ specifying the direction of the horizontal and vertical sides of the window rectangle, and $\vec{w}$ determining the normal of the plane of the window. Unit coordinate vectors $\vec{u}, \vec{v}, \vec{w}$ are obviously

not independent, because each of them is perpendicular to the other two, thus that dependence has also to be taken care of during the setting of camera parameters. To ease the parameter setting phase, instead of specifying the coordinate vector triple, two almost independent vectors are used for the definition of the orientation, which are the normal vector to the plane of the window, called the **view plane normal**, or $\vec{vpn}$ for short, and a so-called **view up vector**, or $\vec{vup}$, whose component that is perpendicular to the normal and is in the plane of $\vec{vpn}$ and $\vec{vup}$ defines the direction of the vertical edge of the window. There is a slight dependence between them, since they should not be parallel, that is, it must always hold that $\vec{vup} \times \vec{vpn} \neq 0$. The $\vec{u}, \vec{v}, \vec{w}$ coordinate vectors can easily be calculated from the view plane normal and the view up vectors:

$$\vec{w} = \frac{\vec{vpn}}{|\vec{vpn}|}, \qquad \vec{u} = \frac{\vec{w} \times \vec{vup}}{|\vec{w} \times \vec{vup}|}, \qquad \vec{v} = \vec{u} \times \vec{w}. \qquad (5.38)$$

Note that unlike the x, y, z world coordinate system, the u, v, w system has been defined left handed to meet the user's expectations that $\vec{u}$ points to the right, $\vec{v}$ points upwards and $\vec{w}$ points away from the camera located behind the window.

- **Size of the window**. The length of the edges of the window rectangle are defined by two positive numbers, the width by $wwidth$, the height by $wheight$. Photographic operations, such as zooming in and out, can be realized by proper control of the size of the window. To avoid distortions, the width/height ratio has to be equal to width/height ratio of the viewport on the screen.

- **Type of projection**. The image is the projection of the virtual world onto the window. Two different types of projection are usually used in computer graphics, the **parallel projection** (if the projectors are parallel), and the **perspective projection** (if all the projectors go through a given point, called the center of projection). Parallel projections are further classified into **orthographic** and **oblique** projections depending on whether or not the projectors are perpendicular to the plane of projection (window plane). The attribute "*oblique*" may also refer to perspective projection if the projector from the center of

the window is not perpendicular to the plane of the window. Oblique projections may cause distortion of the image.

- **Location of the camera or eye**. The camera is placed behind the window in our conceptual model. For perspective projection, the camera position is, in fact, the center of projection, which can be defined by a point $\vec{eye}$ in the u, v, w coordinate system. For parallel projection, the direction of the projectors has to be given by the u, v, w coordinates of the direction vector. Both in parallel and perspective projections the depth coordinate w is required to be negative in order to place the camera "behind" the window. It also makes sense to consider parallel projection as a special perspective projection, when the camera is at an infinite distance from the window.

- **Front and back clipping planes**. According to the conceptual model of taking photos of the virtual world, it is obvious that only those portions of the model which lie in the infinite pyramid defined by the camera as the apex, and the sides of the 3D window (for perspective projection), and in a half-open, infinite parallelepiped (for parallel projection) can affect the photo. These infinite regions are usually limited to a finite frustum of a pyramid, or to a finite parallelepiped respectively, to avoid overflows and also to ease the projection task by eliminating the parts located behind the camera, by defining two clipping planes called the **front clipping plane** and the **back clipping plane**. These planes are parallel with the window and thus have constant w coordinates appropriate for the definition. Thus the front plane is specified by an fp value, meaning the plane $w = fp$, and the back plane is defined by a bp value. Considering the objectives of the clipping planes, their w coordinates have to be greater than the w coordinate of the eye, and $fp < bp$ should also hold.

5.4   Viewing transformation

Image generation involves:

1. the projection of the virtual world onto the window rectangle,

2. the determination of the closest surface at each point (visibility calculation) by depth comparisons if more than one surface can be projected onto the same point in the window, and

3. the placement of the result in the viewport rectangle of the screen.

Obviously, the visibility calculation has to be done prior to the projection of the 3D space onto the 2D window rectangle, since this projection destroys the depth information.

These calculations could also be done in the world coordinate system, but each projection would require the evaluation of the intersection of an arbitrary line and rectangle (window), and the visibility problem would require the determination of the distance of the surface points along the projectors. The large number of multiplications and divisions required by such geometric computations makes the selection of the world coordinate system disadvantageous even if the required calculations can be reduced by the application of the incremental concept, and forces us to look for other coordinate systems where these computations are simple and effective to perform.

In the optimal case the points should be transformed to a coordinate system where X, Y coordinates would represent the pixel location through which the given point is visible, and a third Z coordinate could be used to decide which point is visible, i.e. closest to the eye, if several points could be transformed to the same X, Y pixel. Note that Z is not necessarily proportional to the distance from the eye, it should only be a monotonously increasing function of the distance. The appropriate transformation is also expected to map lines onto lines and planes onto planes, allowing simple representations and linear interpolations during clipping and visibility calculations. Coordinate systems meeting all the above requirements are called **screen coordinate systems**. In a coordinate system of this type, the visibility calculations are simple, since should two or more points have the same X, Y pixel coordinates, then the visible one has the smallest Z coordinate.

From a different perspective, if it has to be decided whether one point will hide another, two comparisons are needed to check whether they project onto the same pixel, that is, whether they have the same X, Y coordinates, and a third comparison must be used to select the closest. The projection is very simple, because the projected point has, in fact, X, Y coordinates due to the definition of the screen space.

For pedagogical reasons, the complete transformation is defined through several intermediate coordinate systems, although eventually it can be accomplished by a single matrix multiplication. For both parallel and perspective cases, the first step of the transformation is to change the coordinate system to u, v, w from x, y, z, but after that there will be differences depending on the projection type.

5.4.1 World to window coordinate system transformation

First, the world is transformed to the u, v, w coordinate system fixed to the center of the window. Since the coordinate vectors $\vec{u}$, $\vec{v}$, $\vec{w}$ and the origin $\vec{vrp}$ are defined in the x, y, z coordinate system, the necessary transformation can be developed based on the results of section 5.2 of this chapter. The matrix formed by the coordinates of the vectors defining the u, v, w coordinate system is:

$$\mathbf{T_{uvw}} = \left[\begin{array}{cccc} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ vrp_x & vrp_y & vrp_z & 1 \end{array} \right], \tag{5.39}$$

$$[x, y, z, 1] = [\alpha, \beta, \gamma, 1] \cdot \mathbf{T_{uvw}}. \tag{5.40}$$

Since $\vec{u}$, $\vec{v}$, $\vec{w}$ are perpendicular vectors, $\mathbf{T_{uvw}}$ is always invertible. Thus, the coordinates of an arbitrary point of the world coordinate system can be expressed in the u, v, w coordinate system as well:

$$[\alpha, \beta, \gamma, 1] = [x, y, z, 1] \cdot \mathbf{T_{uvw}^{-1}}. \tag{5.41}$$

5.4.2 Window to screen coordinate system transformation for parallel projection

Shearing transformation

For oblique transformations, that is when eye_u or eye_v is not zero, the projectors are not perpendicular to the window plane, thus complicating visibility calculations and projection (figure 5.7). This problem can be solved by distortion of the object space, applying a **shearing transformation** in such a way that the non-oblique projection of the distorted objects should provide the same images as the oblique projection of the original scene, and the depth coordinate of the points should not be affected. A general
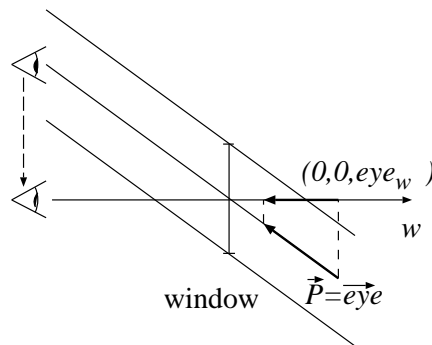
Figure 5.7: Shearing

shearing transformation which does not affect the w coordinate is:

$$\mathbf{T}_{\text{shear}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ s_u & s_v & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.42}$$

The unknown elements, s_u and s_v, can be determined by examining the transformation of the projector $\vec{P} = [eye_u, eye_v, eye_w, 1]$. The transformed projector is expected to be perpendicular to the window and to have depth coordinate eye_w, that is:

$$\vec{P} \cdot \mathbf{T}_{\text{shear}} = [0, 0, eye_w, 1]. \tag{5.43}$$

Using the definition of the shearing transformation, we get:

$$s_u = -\frac{eye_u}{eye_w}, \qquad s_v = -\frac{eye_v}{eye_w}. \tag{5.44}$$

Normalizing transformation

Having accomplished the shearing transformation, the objects for parallel projection are in a space shown in figure 5.8. The subspace which can be projected onto the window is a rectangular box between the front and back clipping plane, having side faces coincident to the edges of the window. To allow uniform treatment, a normalizing transformation can be applied, which maps the box onto a normalized block, called the **canonical view volume**, moving the front clipping plane to 0, the back clipping plane to 1, the other boundaries to $x = 1$, $y = 1$, $x = -1$ and $y = -1$ planes respectively.
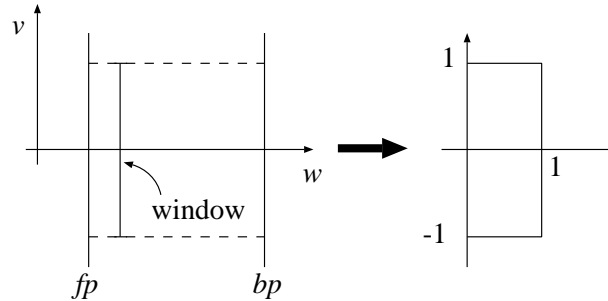
Figure 5.8: Normalizing transformation for parallel projection

The normalizing transformation can also be expressed in matrix form:

$$\mathbf{T_{norm}} = \begin{bmatrix} 2/wwidth & 0 & 0 & 0 \\ 0 & 2/wheight & 0 & 0 \\ 0 & 0 & 1/(bp - fp) & 0 \\ 0 & 0 & -fp/(bp - fp) & 1 \end{bmatrix}. \tag{5.45}$$

The projection in the canonical view volume is very simple, since the projection does not affect the (X, Y) coordinates of an arbitrary point, but only its depth coordinate.

Viewport transformation

The space inside the clipping volume has been projected onto a 2×2 rectangle. Finally, the image has to be placed into the specified viewport of the screen, defined by the center point, (V_x, V_y) and by the horizontal and vertical sizes, V_{sx} and V_{sy}. For parallel projection, the necessary viewport transformation is:

$$
\mathbf{T}_{\mathbf{viewport}} =
\begin{bmatrix}
V_{sx}/2 & 0 & 0 & 0 \\
0 & V_{sy}/2 & 0 & 0 \\
0 & 0 & 1 & 0 \\
V_x & V_y & 0 & 1
\end{bmatrix}.
\tag{5.46}
$$

Summarizing the results, the complete viewing transformation for parallel projection can be generated. The screen space coordinates formed by the (X, Y) pixel addresses and the Z depth value mapped into the range of $[0..1]$ can be determined by the following transformation:

$$
\begin{aligned}
\mathbf{T_V} &= \mathbf{T}_{\mathbf{uvw}}^{-1} \cdot \mathbf{T}_{\mathbf{shear}} \cdot \mathbf{T}_{\mathbf{norm}} \cdot \mathbf{T}_{\mathbf{viewport}}, \\
[X, Y, Z, 1] &= [x, y, z, 1] \cdot \mathbf{T_V}.
\end{aligned}
\tag{5.47}
$$

Matrix $\mathbf{T_V}$, called the **viewing transformation**, is the concatenation of the transformations representing the different steps towards the screen coordinate system. Since $\mathbf{T_V}$ is affine, it obviously meets the requirements of preserving lines and planes, making both the visibility calculation and the projection easy to accomplish.

5.4.3 Window to screen coordinate system transformation for perspective projection

As in the case of parallel projection, objects are first transformed from the world coordinate system to the window, that is u, v, w, coordinate system by applying $\mathbf{T}_{\mathbf{uvw}}^{-1}$.

View-eye transformation

For perspective projection, the center of the u, v, w coordinate system is translated to the camera position without altering the direction of the axes.

Since the camera is defined in the u, v, w coordinate system by a vector $\vec{eye}$, this transformation is a translation by vector $-\vec{eye}$, which can also be expressed by a homogeneous matrix:

$$\mathbf{T_{eye}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -eye_u & -eye_v & -eye_w & 1 \end{bmatrix}. \tag{5.48}$$

Shearing transformation

As for parallel projection, if eye_u or eye_v is not zero, the projector from the center of the window is not perpendicular to the window plane, requiring the distortion of the object space by a **shearing transformation** in such a way that the non-oblique projection of the distorted objects provides the same images as the oblique projection of the original scene and the depth coordinate of the points is not affected. Since the projector from the center of the window ($\vec{P} = [eye_u, eye_v, eye_w, 1]$) is the same as all the projectors for parallel transformation, the shearing transformation matrix will have the same form, independently of the projection type:

$$\mathbf{T_{shear}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -eye_u/eye_w & -eye_v/eye_w & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.49}$$

Normalizing transformation

After shearing transformation the region which can be projected onto the window is a symmetrical, finite frustum of the pyramid in figure 5.9. By normalizing this pyramid, the back clipping plane is moved to 1, and the angle at its apex is set to 90 degrees. This is a simple scaling transformation, with scales S_u, S_v and S_w determined by the consideration that the back clipping plane goes to $w = 1$, and the window goes to the position d which is equal to half the height and half the width of the normalized window:

$$S_u \cdot wwidth/2 = d, \quad S_v \cdot wheight/2 = d, \quad eye_w \cdot S_w = d, \quad S_w \cdot bp = 1 \tag{5.50}$$
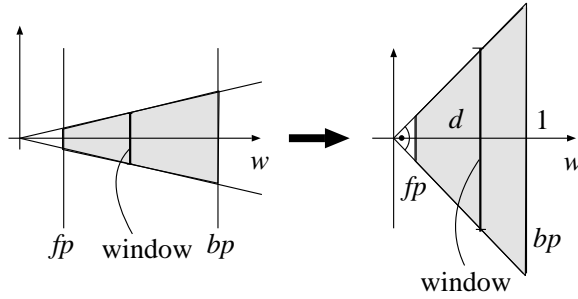
Figure 5.9: Normalizing transformation for perspective projection

Solving these equations and expressing the transformation in a homogeneous matrix form, we get:

$$
\mathbf{T_{norm}} =
\begin{bmatrix}
2 \cdot eye_w/(wwidth \cdot bp) & 0 & 0 & 0 \\
0 & 2 \cdot eye_w/(wheight \cdot bp) & 0 & 0 \\
0 & 0 & 1/bp & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}.
$$
$$(5.51)$$

In the canonical view volume, the central **projection** of a point X_c, Y_c, Z_c onto the window plane is:

$$
X_p = d \cdot \frac{X_c}{Z_c}, \qquad Y_p = d \cdot \frac{Y_c}{Z_c}.
\tag{5.52}
$$

Perspective transformation

The projection and the visibility calculations are more difficult in the canonical view volume for central projection than they are for parallel projection because of the division required by the projection. When calculating visibility, it has to be decided if one point (X_c^1, Y_c^1, Z_c^1) hides another point (X_c^2, Y_c^2, Z_c^2). This involves the check for relations

$$
[X_c^1/Z_c^1, Y_c^1/Z_c^1] = [X_c^2/Z_c^2, Y_c^2/Z_c^2] \quad \text{and} \quad Z_c^1 < Z_c^2
$$

which requires division in a way that the visibility check for parallel projection does not. To avoid division during the visibility calculation, a transformation is needed which transforms the canonical view volume to meet the

requirements of the screen coordinate systems, that is, X and Y coordinates are the pixel addresses in which the point is visible, and Z is a monotonous function of the original distance from the camera (see figure 5.10).
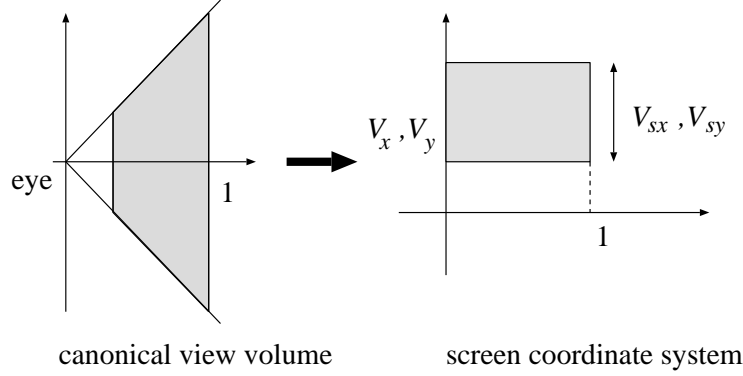

Figure 5.10: Canonical view volume to screen coordinate system transformation

Considering the expectations for the X and Y coordinates:

$$X = \frac{X_c}{Z_c} \cdot \frac{V_{sx}}{2} + V_x, \qquad Y = \frac{Y_c}{Z_c} \cdot \frac{V_{sy}}{2} + V_y. \tag{5.53}$$

The unknown function $Z(Z_c)$ can be determined by forcing the transformation to preserve planes and lines. Suppose a set of points of the canonical view volume are on a plane with the equation:

$$a \cdot X_c + b \cdot Y_c + c \cdot Z_c + d = 0 \tag{5.54}$$

The transformation of this set is also expected to lie in a plane, that is, there are parameters $a', b,' c,' d'$ satisfying the equation of the plane for transformed points:

$$a' \cdot X + b' \cdot Y + c' \cdot Z + d' = 0 \tag{5.55}$$

Inserting formula 5.53 into this plane equation and multiplying both sides by Z_c, we get:

$$a' \cdot \frac{V_{sx}}{2} \cdot X_c + b' \cdot \frac{V_{sy}}{2} \cdot Y_c + c' \cdot Z(Z_c) \cdot Z_c + (a' \cdot V_x + b' \cdot V_y + d') \cdot Z_c = 0 \tag{5.56}$$

Comparing this with equation 5.54, we can conclude that both $Z(Z_c) \cdot Z_c$ and Z_c are linear functions of X_c and Y_c, requiring $Z(Z_c) \cdot Z_c$ to be a linear function of Z_c also. Consequently:

$$Z(Z_c) \cdot Z_c = \alpha \cdot Z_c + \beta \quad \Longrightarrow \quad Z(Z_c) = \alpha + \frac{\beta}{Z_c}. \qquad (5.57)$$

Unknown parameters α and β are set to map the front clipping plane of the canonical view volume ($fp' = fp/bp$) to 0 and the back clipping plane (1) to 1:

$$\alpha \cdot fp' + \beta = 0, \qquad\qquad \alpha \cdot 1 + \beta = 1$$
$$\Downarrow \qquad\qquad\qquad\qquad (5.58)$$
$$\alpha = bp/(bp - fp), \qquad \beta = -fp/(bp - fp)$$

The complete transformation, called the **perspective transformation**, is:

$$X = \frac{X_c}{Z_c} \cdot \frac{V_{sx}}{2} + V_x, \qquad Y = \frac{Y_c}{Z_c} \cdot \frac{V_{sy}}{2} + V_y, \qquad Z = \frac{Z_c \cdot bp - fp}{(bp - fp) \cdot Z_c}. \quad (5.59)$$

Examining equation 5.59, we can see that $X \cdot Z_c$, $Y \cdot Z_c$ and $Z \cdot Z_c$ can be expressed as a linear transformation of X_c, Y_c, Z_c, that is, in homogeneous coordinates $[X_h, Y_h, Z_h, h] = [X \cdot Z_c, Y \cdot Z_c, Z \cdot Z_c, Z_c]$ can be calculated with a single matrix product by $\mathbf{T_{persp}}$:

$$\mathbf{T_{persp}} = \begin{bmatrix} V_{sx}/2 & 0 & 0 & 0 \\ 0 & V_{sy}/2 & 0 & 0 \\ V_x & V_y & bp/(bp - fp) & 1 \\ 0 & 0 & -fp/(bp - fp) & 0 \end{bmatrix}. \qquad (5.60)$$

The complete perspective transformation, involving homogeneous division to get real 3D coordinates, is:

$$[X_h, Y_h, Z_h, h] = [X_c, Y_c, Z_c, 1] \cdot \mathbf{T_{persp}},$$

$$[X, Y, Z, 1] = [\frac{X_h}{h}, \frac{Y_h}{h}, \frac{Z_h}{h}, 1]. \qquad (5.61)$$

The division by coordinate h is meaningful only if $h \neq 0$. Note that the complete transformation is a homogeneous linear transformation which consists of a matrix multiplication and a homogeneous division to convert the homogeneous coordinates back to Cartesian ones.

This is not at all surprising, since one reason for the emergence of projective geometry has been the need to handle central projection somehow by linear means. In fact, the result of equation 5.61 could have been derived easily if it had been realized first that a homogeneous linear transformation would solve the problem (figure 5.10). This transformation would transform the eye onto an ideal point and make the side faces of the viewing pyramid parallel. Using homogeneous coordinates this transformation means that:

$$\mathcal{T} : \quad [0, 0, 0, 1] \mapsto \lambda_1 [0, 0, -1, 0]. \tag{5.62}$$

Multiplicative factor λ_1 indicates that all homogeneous points differing by a scalar factor are equivalent. In addition, the corner points where the side faces and the back clipping plane meet should be mapped onto the corner points of the viewport rectangle on the $Z = 1$ plane and the front clipping plane must be moved to the origin, thus:

$$
\begin{aligned}
\mathcal{T} : \quad [1, 1, 1, 1] &\mapsto \lambda_2 [V_x + V_{sx}/2, V_y + V_{sy}/2, 1, 1], \\
\mathcal{T} : \quad [1, -1, 1, 1] &\mapsto \lambda_3 [V_x + V_{sx}/2, V_y - V_{sy}/2, 1, 1], \\
\mathcal{T} : \quad [-1, 1, 1, 1] &\mapsto \lambda_4 [V_x - V_{sx}/2, V_y + V_{sy}/2, 1, 1], \\
\mathcal{T} : \quad [0, 0, fp', 1] &\mapsto \lambda_5 [V_x, V_y, 0, 1].
\end{aligned}
\tag{5.63}
$$

Transformation $\mathcal{T}$ is defined by a matrix multiplication with $\mathbf{T}_{4 \times 4}$. Its unknown elements can be determined by solving the linear system of equations generated by equations 5.62 and 5.63. The problem is not determinant since the number of equations (20) is one less than the number of variables (21). In fact, it is natural, since scalar multiples of homogeneous matrices are equivalent. By setting λ_2 to 1, however, the problem will be determinant and the resulting matrix will be the same as derived in equation 5.60.

As has been proven, homogeneous transformation preserves linear sets such as lines and planes, thus deriving this transformation from the requirement that it should preserve planes also guaranteed the preservation of lines.

However, when working with finite structures, such as line segments, polygons, convex hulls, etc., homogeneous transformations can cause serious problems if the transformed objects intersect the $h = 0$ hyperplane. (Note that the preservation of convex hulls could be proven for only those cases when the image of transformation has no such intersection.)

To demonstrate this problem and how perspective transformation works, consider an example when $V_x = V_y = 0, V_{sx} = V_{sy} = 2, fp = 0.5, bp = 1$ and

1. Canonical view volume in 3D Euclidean space
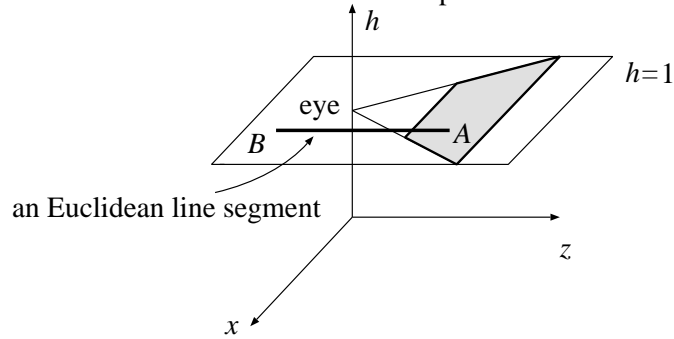
2. After the perspective transformation
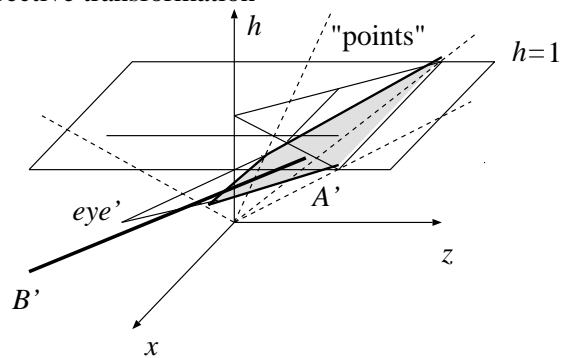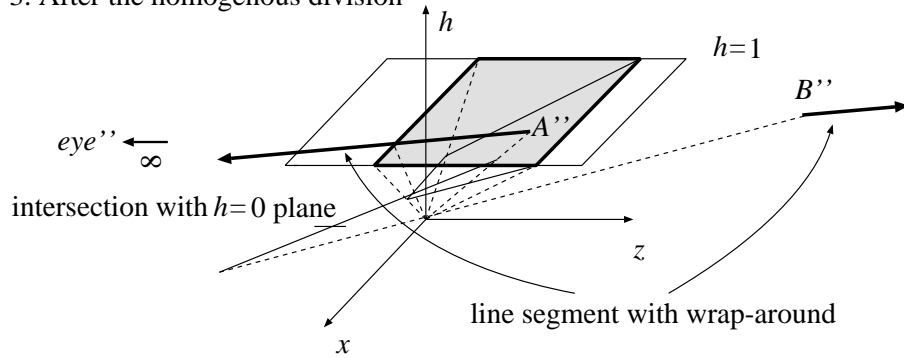
3. After the homogenous division

Figure 5.11: Steps of the perspective transformation and the wrap-around problem

examine what happens with the clipping region and with a line segment defined by endpoints [0.3,0,0.6] and [0.3,0,-0.6] in the Cartesian coordinate system (see figure 5.11). This line segment starts in front of the eye and goes behind it. When the homogeneous representation of this line is transformed by multiplying the perspective transformation matrix, the line will intersect the $h = 0$ plane, since originally it intersects the $Z_c = 0$ plane (which is parallel with the window and contains the eye) and the matrix multiplication sets $h = Z_c$. Recall that the $h = 0$ plane corresponds to the ideal points in the straight model, which have no equivalent in Euclidean geometry.

The conversion of the homogeneous coordinates to Cartesian ones by homogeneous division maps the upper part corresponding to positive h values onto a Euclidean half-line and maps the lower part corresponding to negative h values onto another half-line. This means that the line segment falls into two half-lines, a phenomenon which is usually referred to as the **wrap-around problem**.

Line segments are identified by their two endpoints in computer graphics. If wrap-around phenomena may occur we do not know whether the transformation of the two endpoints really define the new segment, or these are the starting points of two half-lines that form the complement of the Euclidean segment. This is not surprising in projective geometry, since a projective version of a Euclidean line, for example, also includes an ideal point in addition to all affine points, which glues the two "ends" of the line at infinity. From this point of view projective lines are similar (more precisely isomorphic) to circles. As two points on a circle cannot identify an arc unambiguously, two points on a projective line cannot define a segment either without further information. By knowing, however, that the projective line segment does not contain ideal points, this definition is unambiguous.

The elimination of ideal points from the homogeneous representation before homogeneous division obviously solves the problem. Before the homogeneous division, this procedure cuts the objects represented by homogeneous coordinates into two parts corresponding to the positive and negative h values respectively, then projects these parts back to the Cartesian coordinates separately and generates the final representation as the union of the two cases. Recall that a clipping that removes object parts located outside of the viewing pyramid must be accomplished somewhere in the viewing pipeline. The cutting proposed above is worth combining with this clipping step, meaning that the clipping (or at least the so-called depth clip-

ping phase that can remove the vanishing plane which is transformed onto ideal points) must be carried out before homogeneous division. Clipping is accomplished by appropriate algorithms discussed in the next section.

Summarizing the transformation steps of viewing for the perspective case, the complete viewing transformation is:

$$
\begin{aligned}
\mathbf{T_V} &= \mathbf{T_{uvw}^{-1}} \cdot \mathbf{T_{eye}} \cdot \mathbf{T_{shear}} \cdot \mathbf{T_{norm}} \cdot \mathbf{T_{persp}}, \\
[X_h, Y_h, Z_h, h] &= [x, y, z, 1] \cdot \mathbf{T_V}, \\
[X, Y, Z, 1] &= [\frac{X_h}{h}, \frac{Y_h}{h}, \frac{Z_h}{h}, 1].
\end{aligned}
\tag{5.64}
$$

5.5 Clipping

Clipping is responsible for eliminating those parts of the scene which do not project onto the window rectangle, because they are outside the viewing volume. It consists of depth — front and back plane — clipping and clipping at the side faces of the volume. For perspective projection, depth clipping is also necessary to solve the wrap-around problem, because it eliminates the objects in the plane parallel to the window and incident to the eye, which are mapped onto the ideal plane by the perspective transformation.

For parallel projection, depth clipping can be accomplished in any coordinate system before the projection, where the depth information is still available. The selection of the coordinate system in which the clipping is done may depend on efficiency considerations, or more precisely:

1. The geometry of the clipping region has to be simple in the selected coordinate system in order to minimize the number of necessary operations.

2. The transformation to the selected coordinate system from the world coordinate system and from the selected coordinate system to pixel space should involve the minimum number of operations.

Considering the first requirement, for parallel projection, the brick shaped canonical view volume of the normalized eye coordinate system and the screen coordinate system are the best, but, unlike the screen coordinate system, the normalized eye coordinate system requires a new transformation after clipping to get to pixel space. The screen coordinate system thus ranks

as the better option. Similarly, for perspective projection, the pyramid shaped canonical view volume of the normalized eye and the homogeneous coordinate systems require the simplest clipping calculations, but the latter does not require extra transformation before homogeneous division. For side face clipping, the screen coordinate system needs the fewest operations, but separating the depth and side face clipping phases might be disadvantageous for specific hardware realizations. In the next section, the most general case, clipping in homogeneous coordinates, will be discussed. The algorithms for other 3D coordinate systems can be derived from this general case by assuming the homogeneous coordinate h to be constant.

5.5.1 Clipping in homogeneous coordinates

The boundaries of the clipping region can be derived by transforming the requirements of the screen coordinate system to the homogeneous coordinate system. After homogeneous division, in the screen coordinate system the boundaries are $X_{\min} = V_x - V_{sx}/2$, $X_{\max} = V_x + V_{sx}/2$, $Y_{\min} = V_y - V_{sy}/2$ and $Y_{\max} = V_y + V_{sy}/2$. The points internal to the clipping region must satisfy:

$$
\begin{aligned}
X_{\min} &\leq X_h/h \leq X_{\max}, \\
Y_{\min} &\leq Y_h/h \leq Y_{\max}, \\
0 &\leq Z_h/h \leq 1
\end{aligned}
\tag{5.65}
$$

The visible parts of objects defined in an Euclidean world coordinate system must have positive Z_c coordinates in the canonical view coordinate system, that is, they must be in front of the eye. Since multiplication by the perspective transformation matrix sets $h = Z_c$, for visible parts, the fourth homogeneous coordinate must be positive. Adding $h > 0$ to the set of inequalities 5.65 and multiplying both sides by this positive h, an equivalent system of inequalities can be derived as:

$$
\begin{aligned}
X_{\min} \cdot h &\leq X_h \leq X_{\max} \cdot h, \\
Y_{\min} \cdot h &\leq Y_h \leq Y_{\max} \cdot h, \\
0 &\leq Z_h \leq h.
\end{aligned}
\tag{5.66}
$$

Note that inequality $h > 0$ does not explicitly appear in the requirements, since it comes from $0 \leq Z_h \leq h$. Inequality $h > 0$, on the other hand, guarantees that all points are eliminated that are on the $h = 0$ ideal plane, which solves the wrap-around problem.
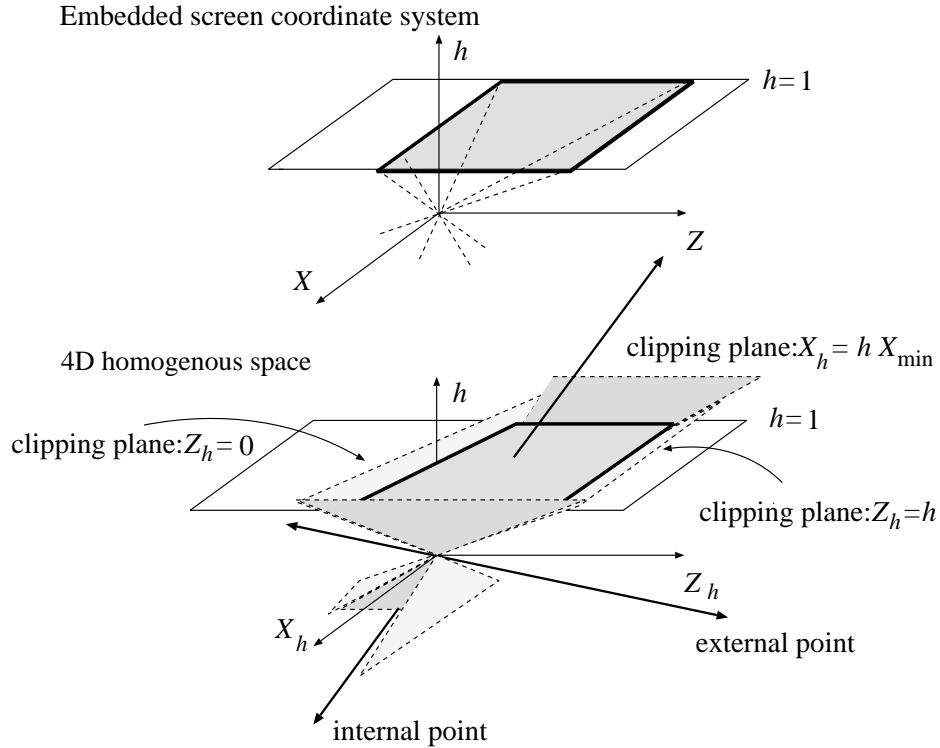
Embedded screen coordinate system

Figure 5.12: Transforming the clipping region back to projective space

Notice that the derivation of the homogeneous form of clipping has been achieved by transforming the clipping box defined in the screen coordinate system back to the projective space represented by homogeneous coordinates (figure 5.12).

When the definition of the clipping region was elaborated, we supposed that the objects are defined in a Cartesian coordinate system and relied on the camera construction discussed in section 5.3. There are fields of computer graphics, however, where none of these is true. Sometimes it is more convenient to define the objects directly in the projective space by homogeneous coordinates. A rational B-spline, for example, can be defined as a non-rational B-spline in homogeneous space, since the homogeneous to Cartesian mapping will carry out the division automatically. When dealing with homogeneous coordinates directly, scalar multiples of the coordinates

are equivalent, thus both positive and negative h regions can contribute to the visible section of the final space. Thus, equation 5.65 must be converted to two system of inequalities, one supposing $h > 0$, the other $h < 0$.

$$
\begin{array}{cc}
\textbf{Case 1: } h > 0 & \textbf{Case 2: } h < 0 \\
X_{\min} \cdot h \leq X_h \leq X_{\max} \cdot h & X_{\min} \cdot h \geq X_h \geq X_{\max} \cdot h \\
Y_{\min} \cdot h \leq Y_h \leq Y_{\max} \cdot h & Y_{\min} \cdot h \geq Y_h \geq Y_{\max} \cdot h \\
0 \leq Z_h \leq h & 0 \geq Z_h \geq h
\end{array}
\tag{5.67}
$$

Clipping must be carried out for the two regions separately. After homogeneous division these two parts will meet in the screen coordinate system.

Even this formulation — which defined a front clipping plane in front of the eye to remove points in the vanishing plane — may not be general enough for systems where the clipping region is independent of the viewing transformation like in PHIGS [ISO90]. In the more general case the image of the clipping box in the homogeneous space may have intersection with the ideal plane, which can cause wrap-around. The basic idea remains the same in the general case; we must get rid of ideal points by some kind of clipping. The interested reader is referred to the detailed discussion of this approach in [Kra89],[Her91].

Now the clipping step is investigated in detail. Let us assume that the clipping region is defined by equation 5.66 (the more general case of equation 5.67 can be handled similarly by carrying out two clipping procedures).

Based on equation 5.66 the clipping of points is very simple, since their homogeneous coordinates must be examined to see if they satisfy all the equations. For more complex primitives, such as line segments and planar polygons, the intersection of the primitive and the planes bounding the clipping region has to be calculated, and that part of the primitive should be preserved where all points satisfy equation 5.66. The intersection calculation of bounding planes with line segments and planar polygons requires the solution of a linear equation involving multiplications and divisions. The case when there is no intersection happens when the solution for a parameter is outside the range of the primitive. The number of divisions and multiplications necessary can be reduced by eliminating those primitive-plane intersection calculations which do not provide intersection, assuming that there is a simple way to decide which these are. Clipping algorithms contain special geometric considerations to decide if there might be an intersection without solving the linear equation.

Clipping of line segments

One of the simplest algorithms for clipping line segments with fewer intersection calculations is the 3D extension of the Cohen and Sutherland clipping algorithm.

Each bounding plane of the clipping region divides the 3D space into two half-spaces. Points in 3D space can be characterized by a 6-bit code, where each bit corresponds to a respective plane defining whether the given point and the convex view volume are on opposite sides of the plane by 1 (or true) value, or on the same side of the plane, by 0 (or false) value. Formally the code bits $C[0] \ldots C[5]$ of a point are defined by:

$$C[0] = \begin{cases} 1 \text{ if } X_h < X_{\min} \cdot h \\ 0 \text{ otherwise} \end{cases} \qquad C[1] = \begin{cases} 1 \text{ if } X_h > X_{\max} \cdot h \\ 0 \text{ otherwise} \end{cases}$$

$$C[2] = \begin{cases} 1 \text{ if } Y_h < Y_{\min} \cdot h \\ 0 \text{ otherwise} \end{cases} \qquad C[3] = \begin{cases} 1 \text{ if } Y_h > Y_{\max} \cdot h \\ 0 \text{ otherwise} \end{cases} \qquad (5.68)$$

$$C[4] = \begin{cases} 1 \text{ if } Z_h < 0 \\ 0 \text{ otherwise} \end{cases} \qquad C[5] = \begin{cases} 1 \text{ if } Z_h > h \\ 0 \text{ otherwise} \end{cases}$$

Obviously, points coded by 000000 have to be preserved, while all other codes correspond to regions outside the view volume (figure 5.13).
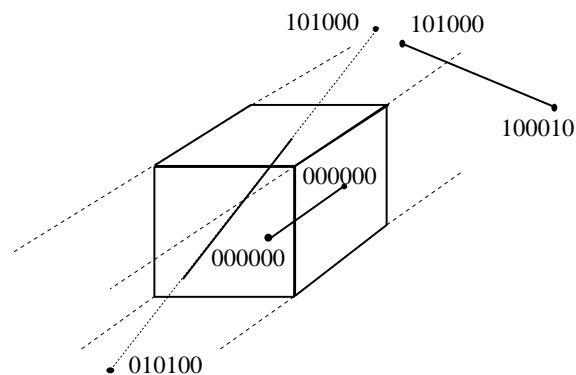

Figure 5.13: Clipping of line segments

Let the codes of the two endpoints of a line segment be C_1 and C_2 respectively. If both C_1 and C_2 are zero, the endpoints, as well as all inner

points of the line segment, are inside the view volume, thus the whole line segment has to be preserved by clipping. If the corresponding bits of both C_1 and C_2 are non-zero at some position, then the endpoints, and the inner points too, are on the same side of the respective bounding plane, external to the view volume, thus requiring the whole line segment to be eliminated by clipping. These are the trivial cases where clipping can be accomplished without any intersection calculation.

If this is not the case — that is if at least one bit pair in the two codes are not equal, and for those bits where they are the same, they have a value of 0, then the intersection between the line and that plane which corresponds to the bit where the two codes are different has to be calculated, and the part of the line segment which is outside must be eliminated by replacing the endpoint having 1 code bit by the intersection point. Let the two endpoints have coordinates $[X_h^{(1)}, Y_h^{(1)}, Z_h^{(1)}, h^{(1)}]$ and $[X_h^{(2)}, Y_h^{(2)}, Z_h^{(2)}, h^{(2)}]$ respectively. The parametric representation of the line segment, supposing parameter range $[0..1]$ for t, is:

$$
\begin{aligned}
X_h(t) &= X_h^{(1)} \cdot t + X_h^{(2)} \cdot (1 - t) \\
Y_h(t) &= Y_h^{(1)} \cdot t + Y_h^{(2)} \cdot (1 - t) \\
Z_h(t) &= Z_h^{(1)} \cdot t + Z_h^{(2)} \cdot (1 - t) \\
h(t) &= h^{(1)} \cdot t + h^{(2)} \cdot (1 - t)
\end{aligned}
$$
(5.69)

Note that this representation expresses the line segment as a linear set spanned by the two endpoints. Special care has to be taken when the h coordinates of the two endpoints have different sign because this means that the linear set contains an ideal point as well.

Now let us consider the intersection of this line segment with a clipping plane (figure 5.14). If, for example, the code bits are different in the first bit corresponding to $X_{\min}$, then the intersection with the plane $X_h = X_{\min} \cdot h$ has to be calculated thus:

$$
X_h^{(1)} \cdot t + X_h^{(2)} \cdot (1 - t) = X_{\min} \cdot (h^{(1)} \cdot t + h^{(2)} \cdot (1 - t)). \tag{5.70}
$$

Solving for parameter t^* of the intersection point, we get:

$$
t^* = \frac{X_{\min} \cdot h^{(2)} - X_h^{(2)}}{X_h^{(1)} - X_h^{(2)} - X_{\min} \cdot (h^{(1)} - h^{(2)})}. \tag{5.71}
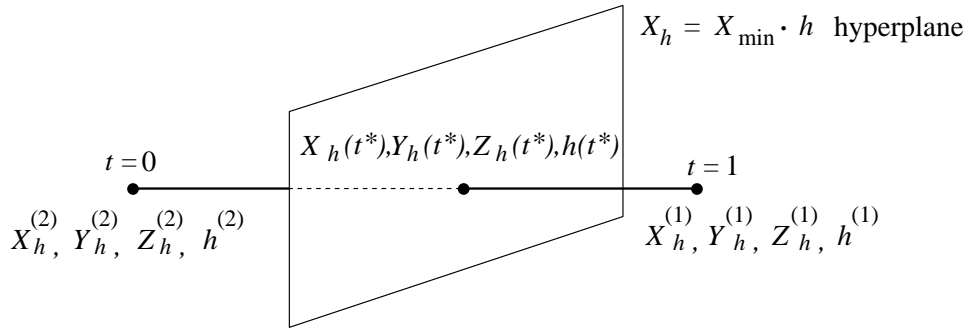$$

Figure 5.14: Clipping by a homogeneous plane

Substituting t^* back to the equation of the line segment, the homogeneous coordinates of the intersection point are $[X_h(t^*), Y_h(t^*), Z_h(t^*), h(t^*)]$. For other bounding planes, the algorithm is similar. The steps discussed can be converted into an algorithm which takes and modifies the two endpoints and returns TRUE if some inner section of the line segment is found, and FALSE if the segment is totally outside the viewing volume, thus:

> **LineClipping**$(P_h^{(1)}, P_h^{(2)})$
>> C_1 = Calculate code bits for $P_h^{(1)}$;
>> C_2 = Calculate code bits for $P_h^{(2)}$;
>> **loop**
>>> **if** $(C_1 = 0 \text{ AND } C_2 = 0)$ **then return** TRUE;   // *Accept*
>>> **if** $(C_1 \ \& \ C_2 \neq 0)$ **then return** FALSE;   // *Reject*
>>> f = Index of clipping face, where bit of C_1 differs from C_2;
>>> P_h^* = Intersection of line $(P_h^{(1)}, P_h^{(2)})$ and plane f;
>>> C^* = Calculate code bits for P_h^*;
>>> **if** $C_1[f] = 1$ **then** $P_h^{(1)} = P_h^*$; $C_1 = C^*$;
>>> **else** $P_h^{(2)} = P_h^*$; $C_2 = C^*$;
>> **endloop**

The previously discussed Cohen–Sutherland algorithm replaces a lot of intersection calculations by simple arithmetics of endpoint codes, increasing the efficiency of clipping, but may still calculate intersections which later

turn out to be outside the clipping region. This means that it is not optimal for the number of calculated intersections. Other algorithms make use of a different compromise in the number of intersection calculations and the complexity of other geometric considerations [CB78], [LB84], [Duv90].

Clipping of polygons

Unfortunately, polygons cannot be clipped by simply clipping the edges, because this may generate false results (see figure 5.15). The core of the problem is the fact that the edges of a polygon can go around the faces of the bounding box, and return through a face different from where they left the inner section, or they may not intersect the faces at all, when the polygon itself encloses or is enclosed by the bounding box.
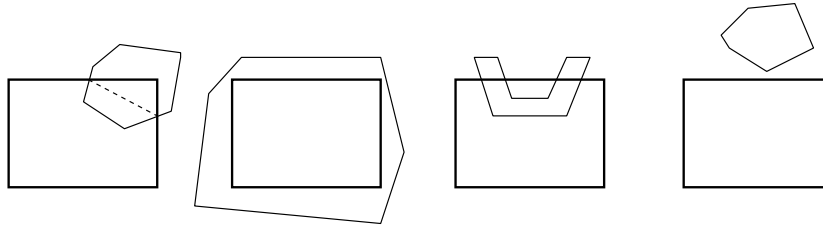
Figure 5.15: Cases of polygon clipping

This problem can be solved if clipping against a bounding box is replaced by six clipping steps to the planes of the faces of the bounding box, as has been proposed for the 2D equivalent of this problem by Hodgman and Sutherland [SH74]. Since planes are infinite objects, polygon edges cannot go around them, and a polygon, clipped against all the six boundary planes, is guaranteed to be inside the view volume.

When clipping against a plane, consecutive vertices have to be examined to determine whether they are on the same side of the plane. If both of them are on the same side of the plane as the region, then the edge is also the edge of the clipped polygon. If both are on the opposite side of the plane from the region, the edge has to be ignored. If one vertex is on the same side and one on the opposite side, the intersection of the edge and the plane has to be calculated, and a new edge formed from that to the point where the polygon returns back through the plane (figure 5.16).
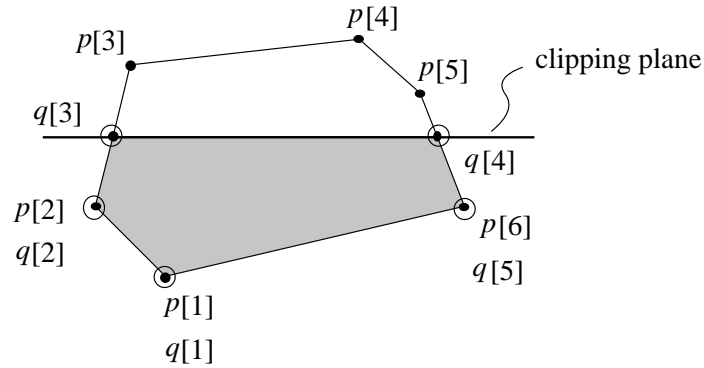
Figure 5.16: Clipping of a polygon against a plane

Suppose the vertices of the polygon are in an array $p[0], \ldots, p[n-1]$, and the clipped polygon is expected in $q[0], \ldots, q[m-1]$, while the number of vertices of the clipped polygon in variable m. The clipping algorithm, using the notation $\oplus$ for modulo n addition, is:

```
m = 0;
for i = 0 to n − 1 do
    if p[i] is inside then {
        q[m++] = p[i];
        if p[i ⊕ 1] is outside then
            q[m++] = Intersection of edge (p[i], p[i ⊕ 1]);
    } else if p[i ⊕ 1] is inside then
            q[m++] = Intersection of edge (p[i], p[i ⊕ 1]);
    endif
endfor
```

Running this algorithm for concave polygons that should fall into several pieces due to clipping (figure 5.17) may result in an even number of edges where no edges should have been generated and the parts that are expected to fall apart are still connected by these even number of boundary lines.

For the correct interpretation of the inside of these polygons, the GKS concept must be used, that is, to test whether a point is inside or outside a polygon, a half-line is extended from the point to infinity and the num-
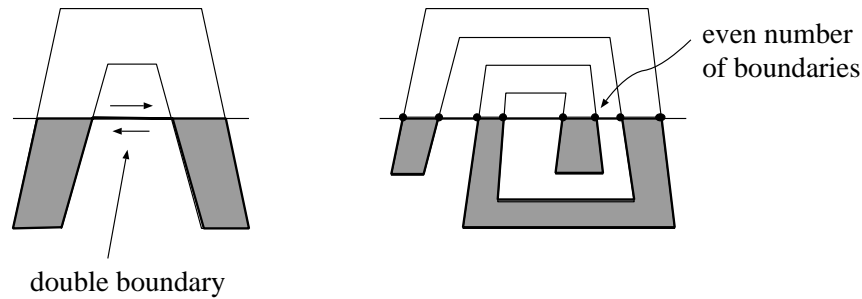
Figure 5.17: Clipping of concave polygons

ber of intersections with polygon boundaries counted. If the line cuts the boundary an odd number of times the point is inside the polygon, if there are even number of intersections the point is outside the polygon. Thus the superficial even number of boundaries connecting the separated parts do not affect the interpretation of inside and outside regions of the polygon.

The idea of Sutherland–Hodgman clipping can be used without modification to clip a polygon against a convex polyhedron defined by planes. A common technique of CAD systems requiring the clipping against an arbitrary convex polyhedron is called **sectioning**, when sections of objects have to be displayed on the screen.

5.6 Viewing pipeline

The discussed phases of transforming the primitives from the world coordinate system to a pixel space are often said to form a so-called **viewing pipeline**. The viewing pipeline is a dataflow model representing the transformations that the primitives have to go through.

Examining figure 5.18, we can see that these viewing pipelines are somehow different from the pipelines discussed by other computer graphics textbooks [FvDFH90], [NS79], because here the screen coordinates contain information about the viewport, in contrast to many authors who define the screen coordinate system as a normalized system independent of the final physical coordinates. For parallel projection, it makes no difference which of the two interpretations is chosen, because the transformations are eventu-
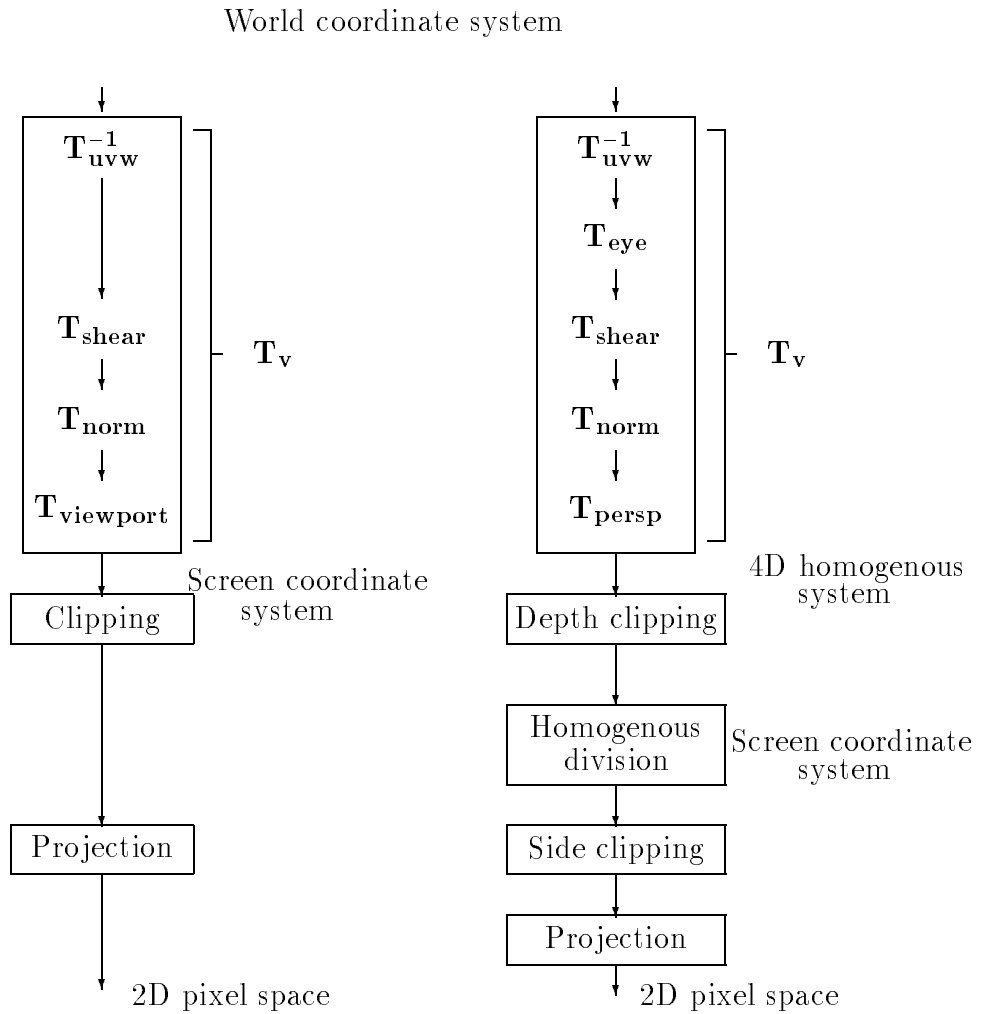
World coordinate system

$$\mathbf{T}^{-1}_{\mathbf{uvw}}$$

$$\mathbf{T}_{\mathbf{shear}}$$

$$\mathbf{T}_{\mathbf{norm}}$$

$$\mathbf{T}_{\mathbf{viewport}}$$

$$\mathbf{T_v}$$

Screen coordinate system

Clipping

Projection

2D pixel space

$$\mathbf{T}^{-1}_{\mathbf{uvw}}$$

$$\mathbf{T}_{\mathbf{eye}}$$

$$\mathbf{T}_{\mathbf{shear}}$$

$$\mathbf{T}_{\mathbf{norm}}$$

$$\mathbf{T}_{\mathbf{persp}}$$

$$\mathbf{T_v}$$

4D homogenous system

Depth clipping

Homogenous division

Screen coordinate system

Side clipping

Projection

2D pixel space

Figure 5.18: Viewing pipeline for parallel and perspective projection

ally concatenated to the same final matrix. For perspective transformation, however, the method discussed here is more efficient, although more difficult to understand, because it does not need an extra transformation to the viewport after the homogeneous division, unlike the approach based on the concept of normalized screen coordinates.

Concerning the coordinate system where the clipping has to be done, figure 5.18 represents only one of many possible alternatives. Nevertheless, this alternative is optimal in terms of the total number of multiplications and divisions required in the clipping and the transformation phases.

At the end of the viewing pipeline the clipped primitives are available in the screen coordinate system which is the primary place of visibility calculations, since here, as has been emphasized, the decision about whether one point hides another requires just three comparisons. Projection is also trivial in the screen coordinate system, since the X and Y coordinates are in fact the projected values.

The angles needed by shading are not invariant to the viewing transformation from the shearing transformation phase. Thus, color computation by the evaluation of the shading equation must be done before this phase. Most commonly, the shading equation is evaluated in the world coordinate system.

5.7 Complexity of transformation and clipping

Let the number of vertices, edges and faces of a polygon mesh model be v, e and f respectively. In order to transform a polygon mesh model from its local coordinate system to the screen coordinate system for parallel projection or to the homogeneous coordinate system for perspective projection, the vector of the coordinates of each vertex must be multiplied by the composite transformation matrix. Thus the time and the space required by this transformation are obviously proportional to the number of vertices, that is, the transformation is an $O(v)$ algorithm.

Clipping may alter the position and the number of vertices of the representation. For wireframe display, line clipping is accomplished, which can return a line with its original or new endpoints, or it can return no line at

all. The time for clipping is $O(e)$, and it returns $2e$ number of points in the worst case. Projection processes these points or the resulting clipped edges independently, thus it is also an $O(e)$ process. For wireframe image synthesis, the complexity of the complete viewing pipeline operation is then $O(v + e)$. According to **Euler's theorem**, for normal solids, it holds that:

$$f + v = e + 2 \tag{5.72}$$

Thus, $e = v + f - 2 > v$ for normal objects, which means that pipeline operation has $O(e)$ complexity.

For shaded image synthesis, the polygonal faces of the objects must be clipped. Let us consider the intersection problem of polygon i having e_i edges and a clipping plane. In the worst case all edges intersect the plane, which can generate e new vertices on the clipping plane. The discussed clipping algorithm connects these new vertices by edges, which results in at most $e_i/2$ new edges. If all the original edges are preserved (partially) by the clipping, then the maximal number of edges of the clipped polygon is $e_i + e_i/2$. Thus an upper bound for the number of edges clipped by the 6 clipping plane is $(3/2)^6 \cdot e_i = \text{const} \cdot e_i$.

Since in the polygon mesh model an edge is adjacent to two faces, an upper bound for the number of points which must be projected is:

$$2 \cdot \text{const} \cdot (e_1 + \ldots + e_f) = 4 \cdot \text{const} \cdot e. \tag{5.73}$$

Hence the pipeline also requires $O(e)$ time for the polygon clipping mode.

In order to increase the efficiency of the pipeline operation, the method of **bounding boxes** can be used. For objects or group of objects, bounding boxes that completely include these objects are defined in the local or in the world coordinate system, and before transforming the objects their bounding boxes are checked whether or not their image is inside the clipping region. If it is outside, then the complete group of objects is rejected without further calculations.

Chapter 6

VISIBILITY CALCULATIONS

In order to be able to calculate the color of a pixel we must know from where the light ray through the pixel comes. Of course, as a pixel has finite dimensions, there can be an infinite number of light rays going into the eye through it. In contrast to this fact, an individual color has to be assigned to each pixel, so it will be assumed, at least in this chapter, that each pixel has a specified point, for example its center, and only a single light ray through this point is to be considered. The origin of the ray — if any — is a point on the surface of an object. The main problem is finding this object. This is a geometric searching problem at discrete positions on the image plane. The problem of finding the visible surface points can be solved in one of two ways. Either the pixels can be taken first and then the objects for the individual pixels. In this case, for each pixel of the image, the object which can be seen in it at the special point is determined; the object which is closest to the eye will be selected from those falling onto the pixel point after projection. Alternatively, the objects can be examined before the pixels. Then for the whole scene the parts of the projected image of the objects which are visible on the screen are determined, and then the result is sampled according to the resolution of the raster image. The first approach can solve the visibility problem only at discrete points and the accuracy of the solution depends on the resolution of the screen. This is why it is called an **image-precision** method, also known as an image-space, approximate, finite-resolution or discrete method. The second

approach handles the visible parts of object projections at the precision of
the object description, which is limited only by the finite precision of floating
point calculations in the computer. The algorithms falling into this class are
categorized as **object-precision** algorithms, alternatively as object-space,
exact, infinite-resolution or continuous methods [SSS74].

The following pseudo-codes give a preliminary comparison to emphasize
the differences between the two main categories of visibility calculation al-
gorithms. An image-precision algorithm typically appears as follows:

> **ImagePrecisionAlgorithm**
> > **do**
> > > select a set P of pixels on the screen;
> > > determine visible objects in P;
> > > **for** each pixel $p \in P$ **do**
> > > > draw the object determined as visible at p;
> > >
> > > **endfor**
> >
> > **while** not all pixels computed
>
> **end**

The set of pixels (P) selected in the outer loop depends on the nature
of the algorithm: it can be a single pixel (ray tracing) or a row of pixels
(scan-line algorithm) or the pixels covered by a given object (z-buffer algo-
rithm). An object-precision algorithm, on the other hand, typically appears
as follows:

> **ObjectPrecisionAlgorithm**
> > determine the set S of visible objects;
> > **for** each object $o \in S$ **do**
> > > **for** each pixel p covered by o **do**
> > > > draw o at p;
> > >
> > > **endfor**
> >
> > **endfor**
>
> **end**

If N, R^2 are the number of objects and the number of pixels respectively,
then an image-precision algorithm always has a lower bound of $\Omega(R^2)$ for
its running time, since every pixel has to be considered at least once. An

object-precision algorithm, on the other hand, has a lower bound of $\Omega(N)$ for its time complexity. But these bounds are very optimistic; the first one does not consider that finding the visible object in a pixel requires more and more time as the number of objects grows. The other does not give any indication of how complicated the objects and hence the final image can be. Unfortunately, we cannot expect our algorithms to reach these lower limits.

In the case of image-space algorithms, in order to complete the visibility calculations in a time period proportional to the number of pixels and independent of the number of objects, we would have to be able to determine the closest object along a ray from the eye in a time period independent of the number of objects. But if we had an algorithm that could do this, this algorithm could, let us say, be used for reporting the smallest number in a non-ordered list within time period independent of the number of list elements, which is theoretically impossible. The only way of speeding this up is by preprocessing the objects into some clever data structure before the calculations but there are still theoretical limits.
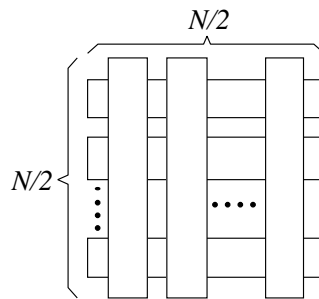
Figure 6.1: Large number of visible parts

In the case of object-space algorithms, let us first consider an extreme example, as shown in figure 6.1. The object scene is a grid consisting of $N/2$ horizontal slabs and $N/2$ vertical slabs in front of the horizontal ones. If the projections of the two groups fall onto each other on the image plane, then the number of the separated visible parts is $\Theta(N^2)$. This simple example shows that an object-precision visibility algorithm with a worst-case running time proportional to the number of objects is impossible, simply because of the potential size of the output.

Since the time spent on visibility calculations is usually overwhelming in 3D rendering, the speed of these algorithms is of great importance. There is no optimal method in either of the two classes (possessing the above-mentioned lower limit speed). This statement, however, holds only if the examinations are performed for the worst case. There are algorithms that have optimal speed in most cases (average case optimal algorithms).

6.1   Simple ray tracing

Perhaps the most straightforward method of finding the point on the surface of an object from where the light ray through a given pixel comes, is to take a half-line starting from the eye and going through (the center of) the pixel, and test it with each object for intersection. Such a ray can be represented by a pair $(\vec{s}, \vec{d})$, where $\vec{s}$ is the starting point of the ray and $\vec{d}$ is its direction vector. The starting point is usually the eye position, while the direction vector is determined by the relative positions of the eye and the actual pixel. Of all the intersection points the one closest to the eye is kept. Following this image-precision approach, we can obtain the simplest ray tracing algorithm:

```
for each pixel p do
    r⃗ = ray from the eye through p;
    visible object = null;
    for each object o do
        if r⃗ intersects o then
            if intersection point is closer than previous ones then
                visible object = o;
            endif
        endif
    endfor
    if visible object ≠ null then
        color of p = color of visible object at intersection point;
    else
        color of p = background color;
    endif
endfor
```

When a ray is to be tested for intersection with objects, each object is taken one by one, hence the algorithm requires $O(R^2 N)$ time (both in worst and average case) to complete the rendering. This is the worst that we can imagine, but the possibilities of this algorithm are so good — we will examine it again in chapter 9 on recursive ray tracing — that despite its slowness ray tracing is popular and it is worth making the effort to accelerate it. The algorithm shown above is the "brute force" form of ray tracing.

The method has a great advantage compared to all the other visible surface algorithms. It works directly in the world coordinate system, it can realize any type of projection, either perspective or parallel, without using transformation matrices and homogeneous division, and finally, clipping is also done automatically (note, however, that if there are many objects falling outside of the viewport then it is worth doing clipping before ray tracing). The first advantage is the most important. A special characteristic of the perspective transformation — including homogeneous division — is that the geometric nature of the object is generally not preserved after the transformation. This means that line segments and polygons, for example, can be represented in the same way as before the transformation, but a sphere will no longer be a sphere. Almost all types of object are sensitive to perspective transformation, and such objects must always be approximated by transformation-insensitive objects, usually by polygons, before the transformation. This leads to loss of geometric information, and adversely affects the quality of the image.

The key problem in ray tracing is to find the intersection between a ray $\vec{r}(\vec{s}, \vec{d})$ and the surface of a geometric object o. Of all the intersection points we are mainly interested in the first intersection along the ray (the one closest the origin of the ray). In order to find the closest one, we usually have to calculate all the intersections between $\vec{r}$ and the surface of o, and then select the one closest to the starting point of $\vec{r}$. During these calculations the following parametric representation of the ray is used:

$$\vec{r}(t) = \vec{s} + t \cdot \vec{d} \quad (t \in [0, \infty)). \tag{6.1}$$

The parameter t refers to the the distance of the actual ray point $\vec{r}(t)$ from the starting point $\vec{s}$. The closest intersection can then be found by comparing the t values corresponding to the intersection points computed.

6.1.1   Intersection with simple geometric primitives

If object o is a sphere, for example, with its center at $\vec{c}$ and of radius R, then the equation of the surface points $\vec{p}$ is:

$$|\vec{p} - \vec{c}| = R \qquad (6.2)$$

where $|\cdot|$ denotes vector length. The condition for intersection between the sphere and a ray $\vec{r}$ is that $\vec{p} = \vec{r}$ for some $\vec{p}$. Substituting the parametric expression 6.1 of ray points for $\vec{p}$ into 6.2, the following quadratic equation is derived with parameter t as the only unknown:

$$(\vec{d})^2 \cdot t^2 + 2 \cdot \vec{d} \cdot (\vec{s} - \vec{c}) \cdot t + (\vec{s} - \vec{c})^2 - R^2 = 0 \qquad (6.3)$$

This equation can be solved using the resolution formula for quadratic equations. It gives zero, one or two different solutions for t, corresponding to the cases of zero, one or two intersection points between the ray and the surface of the sphere, respectively. An intersection point itself can be derived by substituting the value or values of t into expression 6.1 of the ray points. Similar equations to 6.2 can be used for further quadratic primitive surfaces, such as cylinders and cones.

The other type of simple primitive that one often meets is the planar polygon. Since every polygon can be broken down into triangles, the case of a triangle is examined, which is given by its vertices $\vec{a}, \vec{b}$ and $\vec{c}$. One possibility of calculating the intersection point is taking an implicit equation — as in the case of spheres — for the points $\vec{p}$ of the (plane of the) triangle. Such an equation could look like this:

$$((\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})) \cdot (\vec{p} - \vec{a}) = 0 \qquad (6.4)$$

which, in fact, describes the plane containing the triangle. Substituting the expression of the ray into it, a linear equation is constructed for the unknown ray parameter t. This can be solved easily, and always yields a solution, except in cases where the ray is parallel to the plane of the triangle. But there is a further problem. Since equation 6.4 describes not only the points of the triangle, but all the points of the plane containing the triangle, we have to check whether the intersection point is inside the triangle. This leads to further geometric considerations about the intersection point $\vec{p}$. We

can check, for example, that for each side of the triangle, $\vec{p}$ and the third vertex fall onto the same side of it, that is:

$$
\begin{aligned}
((\vec{b} - \vec{a}) \times (\vec{p} - \vec{a})) \cdot ((\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})) &\geq 0, \\
((\vec{c} - \vec{b}) \times (\vec{p} - \vec{b})) \cdot ((\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})) &\geq 0, \\
((\vec{a} - \vec{c}) \times (\vec{p} - \vec{c})) \cdot ((\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})) &\geq 0
\end{aligned}
\tag{6.5}
$$

The point $\vec{p}$ falls into the triangle if and only if all the three inequalities hold.

An alternative approach is to use an explicit expression of the inner points of the triangle. These points can be considered as positive-weighted linear combinations of the three vertices, with a unit sum of weights:

$$
\begin{aligned}
\vec{p}(\alpha, \beta, \gamma) &= \alpha \cdot \vec{a} + \beta \cdot \vec{b} + \gamma \cdot \vec{c}, \\
\alpha, \beta, \gamma &\geq 0, \\
\alpha + \beta + \gamma &= 1
\end{aligned}
\tag{6.6}
$$

The coefficients α, β and γ are also known as the **baricentric coordinates** of point $\vec{p}$ with respect to the spanning vectors $\vec{a}, \vec{b}$ and $\vec{c}$ (as already described in section 5.1). For the intersection with a ray, the condition $\vec{p} = \vec{r}$ must hold, giving a linear equation for the four unknowns α, β, γ and t:

$$
\begin{aligned}
\alpha \cdot \vec{a} + \beta \cdot \vec{b} + \gamma \cdot \vec{c} &= \vec{s} + t \cdot \vec{d}, \\
\alpha + \beta + \gamma &= 1
\end{aligned}
\tag{6.7}
$$

The number of unknowns can be reduced by merging the second equation into the first one. Having solved the merged equation, we have to check whether the resulting intersection point is inside the triangle. In this case, however, we only have to check that $\alpha \geq 0, \beta \geq 0$ and $\gamma \geq 0$.

The two solutions for the case of the triangle represent the two main classes of intersection calculation approaches. In the first case, the surface of the object is given by an implicit equation $F(x, y, z) = 0$ of the spatial coordinates of the surface. In this case, we can always substitute expression 6.1 of the ray into the equation, getting a single equation for the unknown ray parameter t. In the other case, the surface points of the object are given explicitly by a parametric expression $\vec{p} = \vec{p}(u, v)$, where u, v are the surface parameters. In this case, we can always derive an equation system $\vec{p}(u, v) - \vec{r}(t) = \vec{0}$ for the unknowns, u, v and t. In the first case,

the equation is only a single one (although usually non-linear), but objects usually only use a portion of the surface described by the implicit equation and checking that the point is in the part used causes extra difficulties. In the second case, the equation is more complicated (usually a non-linear equation system), but checking the validity of the intersection point requires only comparisons in parameter space.

6.1.2   Intersection with implicit surfaces

In the case where the surface is given by an implicit equation $F(x, y, z) = 0$, the parametric expression 6.1 of the ray can be substituted into it to arrive at the equation $f(t) = F(x(t), y(t), z(t)) = 0$, thus what has to be solved is:

$$f(t) = 0 \qquad\qquad (6.8)$$

This is generally non-linear, and we cannot expect to derive the roots in analytic form, except in special cases.

One more thing should be emphasized here. From all the roots of $f(t)$, we are interested *only* in *its real roots* (complex roots have no geometric meaning). Therefore the problem of finding the real roots will come to the front from now on.

Approximation methods

Generally some **approximation method** must be used in order to compute the roots with any desired accuracy. The problem of approximate solutions to non-linear equations is one of the most extensively studied topics in computational mathematics. We cannot give here more than a collection of related theorems and techniques (mainly taken from the textbook by Demidovich and Maron [DM87]). It will be assumed throughout this section that the function f is continuous and continuously differentiable.

A basic observation is that if $f(a) \cdot f(b) < 0$ for two real numbers a and b, then the interval $[a, b]$ contains at least one root of $f(t)$. This condition of changing the sign is sufficient but not necessary. A counter example is an interval containing an even number of roots. Another counter example is a root where the function has a local minimum or maximum of 0 at the root, that is, the first derivative $f'(t)$ also has a root at the same place as $f(t)$. The reason for the first situation is that the interval contains more

than one root instead of an *isolated* one. The reason for the second case is that the root has a *multiplicity* of more than one. Techniques are known for both isolating the roots and reducing their multiplicity, as we will see later.
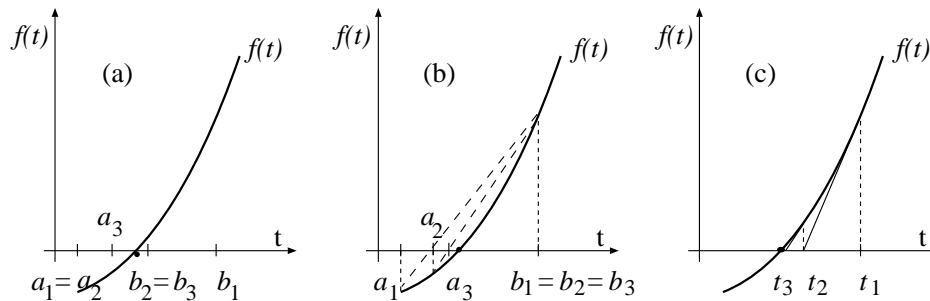

Figure 6.2: Illustrations for the halving (a), chord (b) and Newton's (c) method

If $f(a) \cdot f(b) < 0$ and we know that the interval $[a, b]$ contains exactly one root of $f(t)$, then we can use a number of techniques for approximating this root t^* as closely as desired. Probably the simplest technique is known as the **halving method**. First the interval is divided in half. If $f((a + b)/2) = 0$, then $t^* = (a + b)/2$ and we stop. Otherwise we keep that half, $[a, (a + b)/2]$ or $[(a + b)/2, b]$, at the endpoints of which $f(t)$ has opposite signs. This reduced interval $[a_1, b_1]$ will contain the root. Then this interval is halved in the same way as the original one and the same investigations are made, etc. Continuing this process, we either find the exact value of the root or produce a nested sequence $\{[a_i, b_i]\}$ of intervals of rapidly decreasing width:

$$b_i - a_i = \frac{1}{2^i}(b - a). \tag{6.9}$$

The sequence contracts into a single value in the limiting case $i \to \infty$, which value is the desired root:

$$t^* = \lim_{i \to \infty} a_i = \lim_{i \to \infty} b_i. \tag{6.10}$$

Another simple technique is the **method of chords**, also known as the method of proportional parts. Instead of simply halving the interval $[a, b]$, it is divided at the point where the function would have a root if it were linear (a chord) between $a, f(a)$ and $b, f(b)$. If — without loss of generality

— we assume that $f(a) < 0$ and $f(b) > 0$, then the ratio of the division will be $-f(a)/f(b)$, giving an approximate root value thus:

$$t_1 = a - \frac{f(a)}{f(b) - f(a)}(b - a). \qquad (6.11)$$

If $f(t_1) = 0$, then we stop, otherwise we take the interval $[a, t_1]$ or $[t_1, b]$, depending on that at which endpoints the function $f(t)$ has opposite signs, and produce a second approximation t_2 of the root, etc. The convergence speed of this method is generally faster than that of the halving method.

A more sophisticated technique is **Newton's method**, also known as the method of tangents. It takes more of the local nature of the function into consideration during consecutive approximations. The basic idea is that if we have an approximation t_1 close to the root t^*, and the difference between them is δt, then $f(t^*) = 0$ implies $f(t_1 + \delta t) = 0$. Using the first two terms of Taylor's formula for the latter equation, we get:

$$f(t_1 + \delta t) \approx f(t_1) + f'(t_1) \cdot \delta t = 0 \qquad (6.12)$$

Solving this for δt gives $\delta t \approx -f(t_1)/f'(t_1)$. Adding this to t_1 results in a new (probably closer) approximation of the root t^*. The general scheme of the iteration is:

$$t_{i+1} = t_i - \frac{f(t_i)}{f'(t_i)} \qquad (i = 1, 2, 3, \ldots). \qquad (6.13)$$

The geometric interpretation of the method (see figure 6.2) is that at each approximation t_i the function $f(t)$ is replaced by the tangent line to the curve at $t_i, f(t_i)$ in order to find the next approximation value t_{i+1}. Newton's method is the most rapidly convergent of the three techniques we have looked at so far, but only if the iteration sequence 6.13 is convergent. If we are not careful, it can become divergent. The result can easily depart from the initial interval $[a, b]$ if for some t_i the value of $f'(t_i)$ is much less than that of $f(t_i)$. There are many theorems about "good" initial approximations, from which the approximation sequence is guaranteed to be convergent. One of these is as follows. If $f(a) \cdot f(b) < 0$, and $f'(t)$ and $f''(t)$ are non-zero and preserve signs over $a \le t \le b$, then, proceeding from an initial approximation $t_1 \in [a, b]$ which satisfies $f'(t_1) \cdot f''(t_1) > 0$, it is possible to compute the sole root t^* of $f(t)$ in $[a, b]$ to any degree of accuracy by using

Newton's iteration scheme (6.13). Checking these conditions is by no means a small matter computationally. One possibility is to use interval arithmetic (see section 6.1.3). There are many further approximation methods beyond the three basic ones that we have outlined, but they are beyond the scope of this book.

Reducing the multiplicity of roots of algebraic equations

The function $f(t)$ is algebraic in most practical cases of shape modeling and also in computer graphics. This comes from the fact that surfaces are usually defined by algebraic equations, and the substitution of the linear expression of the ray coordinates also gives an algebraic equation. The term **algebraic** means that the function is a polynomial (rational) function of its variable. Although the function may have a denominator, the problem of solving the equation $f(t) = 0$ is equivalent with the problem of finding the roots of the numerator of $f(t)$ and then checking that the denominator is non-zero at the roots. That is, we can restrict ourselves to equations having the following form:

$$f(t) = a_0 t^n + a_1 t^{n-1} + \ldots + a_n = 0 \qquad (6.14)$$

The fundamental theorem of algebra says that a polynomial of degree n (see equation 6.14 with $a_0 \neq 0$) has exactly n roots, real or complex, provided that each root is counted according to its multiplicity. We say that a root t^* has **multiplicity** m if the following holds:

$$f(t^*) = f'(t^*) = f''(t^*) = \ldots = f^{(m-1)}(t^*) = 0 \quad \text{and} \quad f^{(m)}(t^*) \neq 0 \quad (6.15)$$

We will restrict ourselves to algebraic equations in the rest of the subsection, because this special property can be exploited in many ways.

Multiplicity of roots can cause problems in the approximation of the roots, as we pointed out earlier. Fortunately, any algebraic equation can be **reduced** to another equation of lower or equal degree, which has the same roots, each having a multiplicity of one. If $t_1^*, t_2^*, \ldots, t_k^*$ are the distinct roots of $f(t)$ with multiplicities of $m_1, m_2, \ldots, m_k$, respectively, then the polynomial can be expressed by the following product of terms:

$$f(t) = a_0 (t - t_1^*)^{m_1} (t - t_2^*)^{m_2} \cdots (t - t_k^*)^{m_k}, \quad \text{where} \quad m_1 + m_2 + \ldots + m_k = n.$$
$$(6.16)$$

The first derivative $f'(t)$ can be expressed by the following product:

$$f'(t) = a_0(t - t_1^*)^{m_1-1}(t - t_2^*)^{m_2-1} \cdots (t - t_k^*)^{m_k-1} p(t) \qquad (6.17)$$

where

$$p(t) = m_1(t - t_2^*) \cdots (t - t_k^*) + \ldots + (t - t_1^*) \cdots (t - t_{k-1}^*) m_k. \qquad (6.18)$$

Note that the polynomial $p(t)$ has a non-zero value at each of the roots $t_1^*, t_2^*, \ldots, t_k^*$ of $f(t)$. As a consequence of this, the polynomial:

$$d(t) = a_0(t - t_1^*)^{m_1-1}(t - t_2^*)^{m_2-1} \cdots (t - t_k^*)^{m_k-1} \qquad (6.19)$$

is the greatest common divisor of the polynomials $f(t)$ and $f'(t)$, that is:

$$d(t) = \gcd(f(t), f'(t)). \qquad (6.20)$$

This can be computed using Euclid's algorithm. Dividing $f(t)$ by $d(t)$ yields the following result:

$$g(t) = \frac{f(t)}{d(t)} = (t - t_1^*)(t - t_2^*) \cdots (t - t_k^*) \qquad (6.21)$$

(compare the terms in expression 6.16 of $f(t)$ with those in the expression of $d(t)$). All the roots of $g(t)$ are distinct, have a multiplicity of 1 and coincide with the roots of $f(t)$.

Root isolation

The problem of **root isolation** is to find appropriate disjoint intervals $[a_1, b_1], [a_2, b_2], \ldots, [a_k, b_k]$, each containing exactly one of the distinct real roots $t_1^*, t_2^*, \ldots, t_k^*$, respectively, for the polynomial $f(t)$.

An appropriate first step is to find a finite interval containing all the roots, because it can then be recursively subdivided. Lagrange's theorem helps in this. It states the following about the upper bound R of the positive roots of the equation: Suppose that $a_0 > 0$ in expression 6.14 of the polynomial and a_k ($k \geq 1$) is the first of the negative coefficients (if there is no such coefficient, then $f(t)$ has no positive roots). Then for the upper bound of the positive roots of $f(t)$ we can take the number:

$$R = 1 + \sqrt[k]{\frac{B}{a_0}} \qquad (6.22)$$

where B is the largest absolute value of the negative coefficients of the polynomial $f(t)$. Using a little trick, this single theorem will be enough to give both upper and lower bounds for the positive and negative roots as well. Let us create the following three equations from our original $f(t)$:

$$
\begin{aligned}
f_1(t) &= t^n f\left(\frac{1}{t}\right) = 0, \\
f_2(t) &= f(-t) = 0, \\
f_3(t) &= t^n f\left(-\frac{1}{t}\right) = 0
\end{aligned}
\tag{6.23}
$$

Let the upper bound of their positive roots be R_1, R_2 and R_3, respectively. Then any positive root t^+ and negative root t^- of $f(t)$ will satisfy (R comes from equation 6.22):

$$
\begin{aligned}
\frac{1}{R_1} &\leq t^+ \leq R, \\
-R_2 &\leq t^- \leq -\frac{1}{R_3}.
\end{aligned}
\tag{6.24}
$$

Thus we have at most two finite intervals containing all possible roots. Then we can search for subintervals, each containing exactly one real root. There are a number of theorems of numerical analysis useful for determining the number of real roots in a given interval, such as the one based on Sturm-sequences [Ral65, Ral69] or the Budan–Fourier theorem [DM87]. Instead of reviewing any of these here, a simple method will be shown which is easy to implement and efficient if the degree of the polynomial $f(t)$ is not too large.

The basic observation is that if t_i^* and t_j^* are two distinct roots of of the polynomial $f(t)$ and $t_i^* < t_j^*$, then there is definitely a value $t_i^* < \tau^* < t_j^*$ between them for which $f'(\tau^*) = 0$. This implies that each pair t_i^*, t_{i+1}^* of consecutive roots are separated by a value (or more than one values) τ_i^* ($t_i^* < \tau_i^* < t_{i+1}^*$) for which $f'(\tau_i^*) = 0$ ($1 \leq i \leq k-1$ where k is the number of distinct roots of $f(t)$). This is illustrated in figure 6.3. Note, however, that the contrary is not true: if τ_i^* and τ_j^* are two distinct roots of $f'(t)$ then there is not necessarily a root of $f(t)$ between them. These observations lead to a recursive method:

- Determine the approximate distinct real roots of $f'(t)$. This yields the values $\tau_1^* < \ldots < \tau_{n'}^*$, where $n' < n$ (n is the degree of $f(t)$). Then
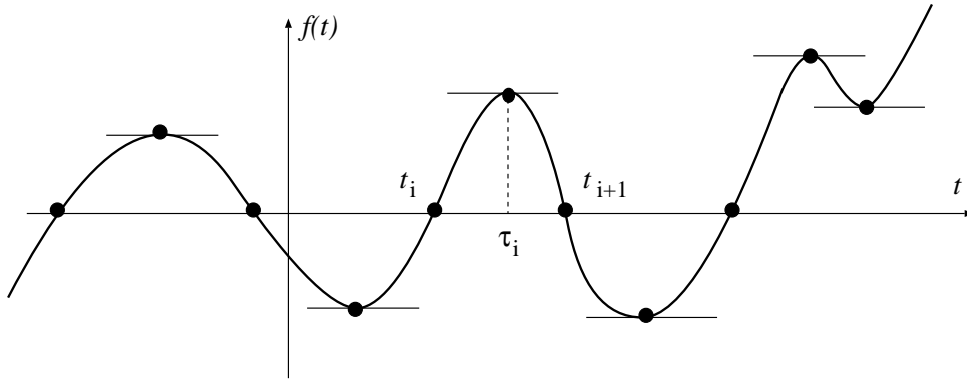
Figure 6.3: Roots isolated by the roots of derivative

each of the $n'+1$ intervals $[-\infty, \tau_1^*], [\tau_1^*, \tau_2^*], \ldots, [\tau_{n'}^*, \infty]$ contains either exactly one root or no roots of $f(t)$. If it is ensured that all the roots of $f(t)$ are of multiplicity 1 (see previous subsection) then it is easy to distinguish between the two cases: if $f(\tau_i^*) \cdot f(\tau_{i+1}^*) < 0$ then the interval $[\tau_i^*, \tau_{i+1}^*]$ contains one root, otherwise it contains no roots. If there is a root in the interval, then an appropriate method can be used to approximate it.

- The approximate distinct real roots of $f'(t)$ can be found recursively. Since the degree of $f'(t)$ is one less than that of $f(t)$ the recursion always terminates.

- At the point where the degree of $f(t)$ becomes 2 (at the bottom of the recursion) the second order equation can be solved easily.

Note that instead of the intervals $[-\infty, \tau_1^*]$ and $[\tau_{n'}^*, \infty]$ the narrower intervals $[-R_2, \tau_1^*]$ and $[\tau_{n'}^*, R]$ can be used, where R_2 and R are defined by equations 6.23 and 6.22.

An example algorithm

As a summary of this section, a possible algorithm is given for approximating all the real roots of a polynomial $f(t)$. It maintains a list L for storing the approximate roots of $f(t)$ and a list L' for storing the approximate roots

of $f'(t)$. The lists are assumed to be sorted in increasing order. The notation $\deg(f(t))$ denotes the degree of the polynomial $f(t)$ (the value of n in expression 6.14):

> **Solve**$(f(t))$
> $L = \{\}$;
>   **if** $\deg(f(t)) < 3$ **then**
>   add roots of $f(t)$ to L // 0, 1 or 2 roots
>   **return** L;
>   **endif**
>   calculate $g(t) = f(t)/\gcd(f(t), f'(t))$;   // eq. 6.21 and 6.20
> $L' = $ **Solve**$(g'(t))$;   // roots of derivative
>   add $-R_2$ and R to L';   // eq. 6.22 and 6.23
> $a = $ first item from L';
>   **while** L' not empty **do**
> $b = $ next item from L';
>   **if** $g(a) \cdot g(b) < 0$ // [a, b] contains one root
> $t = $ approximation of the root in $[a, b]$;
>   add t to L;
>   **endif**
> $a = b$;
>   **endwhile**
>   **return** L;
> **end**

6.1.3 Intersection with explicit surfaces

If we are to find the intersection point between a ray $\vec{r}(t)$ and an explicitly given free-form surface $\vec{s}(u, v)$, then, in fact, the following equation is to be solved:

$$\vec{f}(\vec{x}) = \vec{0}, \qquad (6.25)$$

where $\vec{f}(\vec{x}) = \vec{f}(u, v, t) = \vec{s}(u, v) - \vec{r}(t)$, and the mapping $\vec{f}$ is usually non-linear. We can dispose of the problem of solving a non-linear equation system if we approximate the surface $\vec{s}$ by a finite number of planar polygons and then solve the linear equation systems corresponding to the individual polygons one by one. This method is often used, because it is

straightforward and easy to implement, but if we do not allow such anomalies as jagged contours of smooth surfaces on the picture, then we either have to use a huge number of polygons for the approximation with the snag of having to check all of them for intersection, or we have to use a numerical root-finding method for computing the intersection point within some tolerance.

Newton's method is a classical numerical method for approximating any real root of a non-linear equation system $\vec{f}(\vec{x}) = \vec{0}$. If $[\partial\vec{f}/\partial\vec{x}]$ is the Jacobian matrix of $\vec{f}$ at $\vec{x}$, then the recurrence formula is:

$$\vec{x}_{k+1} = \vec{x}_k - \left[\frac{\partial\vec{f}}{\partial\vec{x}}\right]^{-1} \vec{f}(\vec{x}_k). \qquad (6.26)$$

If our initial guess $\vec{x}_0$ is close enough to a root $\vec{x}^*$, then the sequence $\vec{x}_k$ is convergent, and $\lim_{k\to\infty} \vec{x}_k = \vec{x}^*$. The main problem is how to produce such a good initial guess for each root. A method is needed which always leads to reasonable starting points before performing the iterations. We need, however, computationally performable tests.

One possible method will be introduced in this chapter. The considerations leading to the solution are valid in the n-dimensional real space $\mathbf{R}^n$. For the sake of notational simplicity, the superscript ($\vec{}$) above vector variables will be omitted. They will be reintroduced when returning to our three-dimensional object space.

The method is based on a fundamental theorem of topology: **Schauder's fixpoint theorem** [Sch30, KKM29]. It states that if $X \subset \mathbf{R}^n$ is a convex and compact set and $g\colon \mathbf{R}^n \to \mathbf{R}^n$ is a continuous mapping, then $g(X) \subseteq X$ implies that g has a fixed point $\mathbf{x} \in X$ (that is for which $g(\mathbf{x}) = \mathbf{x}$). Let the mapping g be defined as:

$$g(\mathbf{x}) = \mathbf{x} - \mathbf{Y}f(\mathbf{x}), \qquad (6.27)$$

where $\mathbf{Y}$ is a non-singular $n{\times}n$ matrix. Then, as a consequence of Schauder's theorem, $g(X) \subseteq X$ implies that there is a point $\mathbf{x}^* \in X$ for which:

$$g(\mathbf{x}^*) = \mathbf{x}^* - \mathbf{Y}f(\mathbf{x}^*) = \mathbf{x}^*. \qquad (6.28)$$

Since $\mathbf{Y}$ is non-singular, it implies that $f(\mathbf{x}^*) = 0$. In other words, if $g(X) \subseteq X$, then there is *at least one* solution to $f(\mathbf{x}^*) = 0$ in X. Another important property of the mapping g is that if $\mathbf{x}^* \in X$ is such a root of f, then $g(\mathbf{x}^*) \in X$. This is so because if $f(\mathbf{x}^*) = 0$ then $g(\mathbf{x}^*) = \mathbf{x}^* \in X$. Thus we have a test for the existence of roots of f in a given set X. It is based on the comparison of the set X and its image $g(X)$:

- if $g(X) \subseteq X$ then the answer is positive, that is, X contains *at least one* root

- if $g(X) \cap X = \emptyset$ then the answer is negative, that is, X contains *no* roots, since if it contained one, then this root would also be contained by $g(X)$, but this would be a contradiction

- if none of the above two conditions holds then the answer is neither positive nor negative; in this latter case, however, the set X can be divided into two or more subsets and these smaller pieces can be examined similarly, leading to a recursive algorithm

An important question, if one intends to use this test, is that how the image $g(X)$ and its intersection with X can be computed. Another important problem, if the test gives a positive answer for X, is to decide where to start the Newton-iteration from. A numerical technique, called *interval arithmetic*, gives a possible solution to the first problem. We will survey it here. What it offers is its simplicity, but the price we have to pay is that we never get more than rough estimations for the ranges of mappings. The second problem will be solved by an interval arithmetic based *modification* of the Newton-iteration scheme.

Interval arithmetic

A branch of numerical analysis, called *interval analysis*, basically deals with real intervals, vectors of real intervals, and mappings from and into such objects. Moore's textbook [Moo66] gives a good introduction to it. Our overview contains only those results, which are relevant from the point of view of our problem. Interval objects will be denoted by capital letters.

Let us start with algebraic operations on intervals (addition, subtraction, multiplication and division). Generally, if a binary operation $\circ$ is to be extended to work on two real intervals $X_1 = [a_1, b_1]$ and $X_2 = [a_2, b_2]$, then the rule is:

$$X_1 \circ X_2 = \{x_1 \circ x_2 \mid x_1 \in X_1 \text{ and } x_2 \in X_2\} \tag{6.29}$$

that is, the resulting interval should contain the results coming from all the possible pairings. In the case of subtraction, for example, $X_1 - X_2 = [a_1 - b_2, b_1 - a_2]$. Such an **interval extension** of an operation is *inclusion monotonic*, that is, if $X_1' \subset X_1$ then $X_1' \circ X_2 \subset X_1 \circ X_2$. Based on these operations, the interval extension of an algebraic function can easily be derived by substituting each of its operations by the corresponding interval extension. The (inclusion monotonic) interval extension of a function $f(x)$ will be denoted by $F(X)$. If $f(x)$ is a multidimensional mapping (where x is a vector) then $F(X)$ operates on vectors of intervals called *interval vectors*. The interval extension of a linear mapping can be represented by an *interval matrix* (matrix of intervals).

An interesting fact is that the Lagrangean mean-value theorem extends to the interval extension of functions (although it does not extend to ordinary vector-vector functions). It implies that if f is a continuously differentiable mapping, and F is its interval extension, then for all $\mathbf{x}, \mathbf{y} \in X$:

$$f(\mathbf{x}) - f(\mathbf{y}) \in F'(X)(\mathbf{x} - \mathbf{y}), \tag{6.30}$$

where X is an interval vector (box), $\mathbf{x}, \mathbf{y}$ are real vectors, and F' is the interval extension of the Jacobian matrix of f.

Let us now see some useful definitions. If $X = [a, b]$ is a real interval, then its absolute value, width and middle are defined as:

$$
\begin{aligned}
|X| &= \max(|a|, |b|) &&\text{(absolute value)}, \\
\mathrm{w}\,(X) &= b - a &&\text{(width)}, \\
\mathrm{m}\,(X) &= (a + b)/2 &&\text{(middle)}
\end{aligned}
\tag{6.31}
$$

If $X = (X_1, \ldots, X_n)$ is an interval vector, then its respective vector norm, width and middle vector are defined as:

$$
\begin{aligned}
|X| &= \max_i \{|X_i|\}, \\
\mathrm{w}\,(X) &= \max_i \{\mathrm{w}\,(X_i)\}, \\
\mathrm{m}\,(X) &= (\mathrm{m}\,(X_1), \ldots, \mathrm{m}\,(X_n))
\end{aligned}
\tag{6.32}
$$

For an interval matrix $\mathbf{A} = [A_{ij}]$ the row norm and middle matrix are defined as:

$$\|\mathbf{A}\| = \max_i \left\{ \sum_{j=1}^{n} |A_{ij}| \right\},$$
$$\mathrm{m}(\mathbf{A}) = [\mathrm{m}(A_{ij})] \qquad (6.33)$$

The above defined norm for interval matrices is very useful. We will use the following corollary of this definition later: it can be derived from the definitions [Moo77] that, for any interval matrix $\mathbf{A}$ and interval vector X:

$$\mathrm{w}(\mathbf{A}(X - \mathrm{m}(X))) \leq \|\mathbf{A}\| \cdot \mathrm{w}(X). \qquad (6.34)$$

That is, we can estimate the width of the interval vector containing all the possible images of an interval vector $(X - \mathrm{m}(X))$ if transformed by any of the linear transformations contained in a bundle of matrices (interval matrix $\mathbf{A}$), and we can do this by simple calculations. Note, however, that this inequality can be used only for a special class of interval vectors (origin centered boxes).

Interval arithmetic and the Newton-iteration

We are now in position to perform the test $g(X) \subseteq X$ (equation 6.27) in order to check whether X contains a root (provided that X is a rectangular box): if the interval extension of $g(\mathbf{x})$ is $G(X)$, then $g(X) \subseteq G(X)$, and hence $G(X) \subseteq X$ implies $g(X) \subseteq X$.

Now the question is the following: provided that X contains a root, is the Newton-iteration convergent from any point of X? Another question is that how many roots are in X: only one (a unique root) or more than one? Although it is also possible to answer these questions based on interval arithmetic, the interested reader is referred to Toth's article [Tot85] about this subject. We will present here another method which can be called *an interval version* of the Newton-iteration, first published by Moore [Moo77]. In fact, Toth's work is also based on this method.

The goal of the following argument will be to create an iteration formula, based on the Newton-iteration, which produces a *nested sequence* of interval vectors:

$$X \supset X_1 \supset X_2 \supset \ldots \qquad (6.35)$$

converging to the *unique* solution $\mathbf{x}^* \in X$ if it exists. A test scheme suitable for checking in advance whether a unique $\mathbf{x}^*$ exists will also be provided.

Based on the interval extension $G(X)$ of the mapping $g(\mathbf{x})$ (equation 6.27), consider now the following iteration scheme:

$$X_{k+1} = G(X_k) \quad \text{where } X_0 = X. \tag{6.36}$$

We know that if $G(X) \subseteq X$ then there is *at least one* root $\mathbf{x}^*$ of f in X. It is also sure that for each such $\mathbf{x}^*$, $\mathbf{x}^* \in X_k$ (for all $k \geq 0$), that is, the sequence of interval boxes contains each root. If, furthermore, there exists a positive real number $r < 1$ so that $\mathrm{w}\,(X_{k+1}) \leq r \cdot \mathrm{w}\,(X_k)$ for all $k \geq 0$, then $\lim_{k \to \infty} \mathrm{w}\,(X_k) = 0$, that is, the sequence of interval vectors contracts onto a *single* point. This implies that if the above conditions hold then X contains a *unique* solution $\mathbf{x}^*$ and iteration 6.36 converges to $\mathbf{x}^*$. How can the existence of such a number r (the "contraction factor") be verified in advance?

Inequality 6.34 is suitable for estimating the width of an interval vector resulting from (the interval extension of) a *linear mapping* performed on a *symmetric* interval vector. In order to exploit this inequality, the mapping should be made linear and the interval vector should be made symmetric. Let the expression of mapping g be rewritten as:

$$g(\mathbf{x}) = \mathbf{x} - \mathbf{Y}\,(f(\mathrm{m}\,(X)) + f(\mathbf{x}) - f(\mathrm{m}\,(X))) \tag{6.37}$$

where X can be any interval vector. Following from the Lagrangean mean-value theorem:

$$g(\mathbf{x}) \in \mathbf{x} - \mathbf{Y} f(\mathrm{m}\,(X)) - \mathbf{Y} F'(X)\,(\mathbf{x} - \mathrm{m}\,(X)) \tag{6.38}$$

provided that $\mathbf{x} \in X$. Following from this, the interval extension of g will satisfy (decomposing the right-hand side into a real and an interval term):

$$G(X) \subseteq \mathrm{m}\,(X) - \mathbf{Y} f(\mathrm{m}\,(X)) + [\mathbf{1} - \mathbf{Y} F'(X)]\,(X - \mathrm{m}\,(X)) \tag{6.39}$$

where $\mathbf{1}$ is the unit matrix. Note that the interval mapping on the right-hand side is a linear mapping performed on a symmetric interval vector. Applying now inequality 6.34 (and because $\mathrm{w}\,(X - \mathrm{m}\,(X)) = \mathrm{w}\,(X)$):

$$\mathrm{w}\,(G(X)) \leq \|\mathbf{1} - \mathbf{Y} F'(X)\| \cdot \mathrm{w}\,(X) \tag{6.40}$$

that is, checking whether iteration 6.36 is convergent has become possible. One question is still open: how should the matrix $\mathbf{Y}$ be chosen. Since the structure of the mapping g (equation 6.27) is similar to that of the Newton-step (equation 6.26 with $Y = [\partial \vec{f}/\partial \vec{x}]^{-1}$), intuition tells that $\mathbf{Y}$ should be related to the inverse Jacobian matrix of f (hoping that the convergence speed of the iteration can then be as high as that of the Newton-iteration). Taking the inverse middle of the interval Jacobian $F'(X)$ seems to be a good choice.

In fact, Moore [Moo77] introduced the mapping on the right-hand side of equation 6.39 as a special case of a mapping which he called the **Krawczyk operator**. Let us introduce it for notational simplicity:

$$K(X, \mathbf{y}, \mathbf{Y}) = \mathbf{y} - \mathbf{Y} f(\mathbf{y}) + [\mathbf{1} - \mathbf{Y} F'(X)](X - \mathbf{y}), \qquad (6.41)$$

where X is an interval vector, $\mathbf{y} \in X$ is a real vector, $\mathbf{Y}$ is a non-singular real matrix and f is assumed to be continuously differentiable. The following two properties of this mapping are no more surprising. The first is that if $K(X, \mathbf{y}, \mathbf{Y}) \subseteq X$ for some $\mathbf{y} \in X$, then there exists an $\mathbf{x} \in X$ for which $f(\mathbf{x}) = 0$. The second property is that if $\mathbf{x}^*$ is such a root with $f(\mathbf{x}^*) = 0$, then $\mathbf{x}^* \in K(X, \mathbf{y}, \mathbf{Y})$.

We are now ready to obtain the interval version of Newton's iteration scheme in terms of the Krawczyk operator. Note that this scheme is no else but iteration 6.36 modified so that detecting whether it contracts onto a single point become possible. Setting

$$
\begin{aligned}
X_0 &= X, \\
\mathbf{Y}_0 &= [\mathrm{m}\,(F'(X_0))]^{-1}, \\
r_i &= \|\mathbf{1} - \mathbf{Y}_i F'(X_i)\|
\end{aligned}
\qquad (6.42)
$$

the iteration is defined as follows:

$$X_{i+1} = K(X_i, \mathrm{m}\,(X_i), \mathbf{Y}_i) \cap X_i, \qquad (6.43)$$

$$
\mathbf{Y}_{i+1} = \begin{cases} [\mathrm{m}\,(F'(X_{i+1}))]^{-1}, & \text{if } r_{i+1} \le r_i; \\[2mm] \mathbf{Y}_i, & \text{otherwise} \end{cases}
$$

The initial condition that should be checked before starting the iteration is:

$$K(X_0, \mathrm{m}\,(X_0), \mathbf{Y}_0) \subseteq X_0 \quad \text{and} \quad r_0 < 1 \qquad (6.44)$$

If these two conditions hold, then iteration 6.43 will produce a sequence of *nested* interval boxes converging to the unique solution $\mathbf{x}^* \in X$ of the equation system $f(\mathbf{x}) = 0$.

Let us return to our original problem of finding the intersection point (or all the intersection points) between a ray $\vec{r}(t)$ and an explicitly given surface $\vec{s}(u, v)$. Setting $\vec{f}(\vec{x}) = \vec{f}(u, v, t) = \vec{s}(u, v) - \vec{r}(t)$, the domain X where we have to find all the roots is bounded by some minimum and maximum values of u, v and t respectively. The basic idea of a possible algorithm is that we first check if initial condition 6.44 holds for X. If it does, then we start the iteration process, otherwise we subdivide X into smaller pieces and try to solve the problem on these. The algorithm maintains a list L for storing the approximate roots of $\vec{f}(\vec{x})$ and a list C for storing the candidate interval boxes which may contain solutions:

$$
\begin{aligned}
&C = \{X\}; &&\textit{// candidate list} \\
&L = \{\}; &&\textit{// solution list} \\
&\textbf{while } C \text{ not empty } \textbf{do} \\
&\qquad X_0 = \text{next item on } C; \\
&\qquad \textbf{if } \text{condition 6.44 holds for } X_0 \textbf{ then} \\
&\qquad\qquad \text{perform iteration 6.43 until w}(X_k) \text{ is small enough;} \\
&\qquad\qquad \text{add m}(X_k) \text{ to } L; \\
&\qquad \textbf{else if } \text{w}(X_0) \text{ is not too small } \textbf{then} \\
&\qquad\qquad \text{subdivide } X_0 \text{ into pieces } X_1, \ldots, X_s; \\
&\qquad\qquad \text{add } X_1, \ldots, X_s \text{ to } C; \\
&\qquad \textbf{endif} \\
&\textbf{endwhile}
\end{aligned}
$$

6.1.4   Intersection with compound objects

In constructive solid geometry (CSG) (see subsection 1.6.2) compound objects are given by set operations $(\cup, \cap, \setminus)$ performed on primitive geometric objects such as blocks, spheres, cylinders, cones or even halfspaces bounded by non-linear surfaces. The representation of CSG objects is usually a binary tree with the set operations in its internal nodes and the primitive objects in the leaf nodes. The root of the tree corresponds to the compound object, and its two children represent less complicated objects. If the tree

possesses only a single leaf (and no internal nodes), then the intersection
calculation poses no problem; we have only to compute the intersection be-
tween the ray and a primitive object. On the other hand, if two objects
are combined by a single set operation, and all the intersection points are
known to be on the surface of the two objects, then, considering the opera-
tion, one can easily decide whether any intersection point is on the surface
of the resulting object. For example, if one of the intersection points on the
first object is contained in the interior of the second one, and the combined
object is the union ($\cup$) of the two, then the intersection point is not on its
surface — it is internal to it — hence it can be discarded. Similar argu-
ments can be made for any of the set operations and the possible in/out/on
relationships between a point and an object.

These considerations lead us to a simple **divide-and-conquer** approach:
if the tree has only a single leaf, then the intersection points between the
ray and the primitive object are easily calculated, otherwise — when the
root of the tree is an internal node — the intersection points are recursively
calculated for the left child of the root, taking this child node as the root,
and then the same is done with the right child of the root, and finally the
two sets of intersection points are combined according the set operation at
the root.
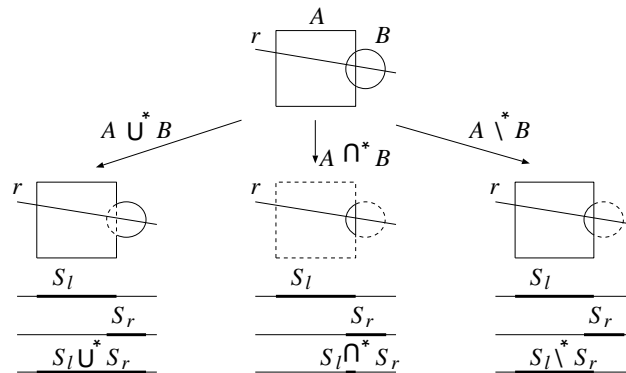
Figure 6.4: Ray spans and their combinations

A slight modification of this approach will help us in considering **regu-
larized set operations** in ray-object intersection calculations. Recall that
it was necessary to introduce regularized set operations in solid modeling

in order to avoid possible anomalies resulting from an operation (see subsection 1.6.1 and figure 1.5). That is, the problem is to find the closest intersection point between a ray and a compound object, provided that the object is built by the use of regularized set operations. Instead of the isolated ray-surface intersection points, we had better deal with line segments resulting from the intersection of the ray and the solid object (more precisely, the closure of the object is to be considered, which is the complement of its exterior). The sequence of consecutive ray segments corresponding to an object will be called a **ray span**. If we take a look at figure 6.4, then we will see how the two ray spans calculated for the two child objects of a node can be combined by means of the set operation of the node. In fact, the result of the combination of the left span S_l and the right span S_r is $S_l \circ^* S_r$, where $\circ^*$ is the set operation ($\cup^*, \cap^*$ or $\backslash^*$). If we really implement the operation $\circ^*$ in the regularized way, then the result will be valid for regularized set operations. This means practically that all segments in a ray span must form a closed set with positive length. There are three cases when regularization takes place. The first is when the result span $S_l \circ^* S_r$ contains an isolated point ($\circ^*$ is $\cap^*$). This point has to be omitted because it would belong to a dangling face, edge or vertex. The second case is when the span contains two consecutive segments, and the endpoint of the first one coincides with the starting point of the second one ($\circ^*$ is $\cup^*$). The two segments have to be merged into one and the double point omitted, because it would belong to a face, edge or vertex (walled-up) in the interior of a solid object. Finally, the third case is when a segment becomes open, that is when one of its endpoints is missing ($\circ^*$ is $\backslash^*$). The segment has to be closed by an endpoint. The algorithm based on the concepts sketched in this subsection is the following:

CSGIntersec(ray, node)
    if node is compound **then**
      left span = **CSGIntersec**(ray, left child of node);
      right span = **CSGIntersec**(ray, right child of node);
      return CSGCombine(left span, right span, operation);
    else (node is a primitive object)
      return PrimitiveIntersec(ray, node);
    endif
end

The intersection point that we are looking for will appear as the starting point of the first segment of the span.

6.2   Back-face culling

It will be assumed in this and all the consecutive sections of this chapter that objects are transformed into the screen coordinate system, and that in the case of perspective projection the homogeneous division has also been performed. This means that objects have to be projected orthographically onto the image plane spanned by the coordinate axes X, Y, and the coordinate axis Z coincides with the direction of view.
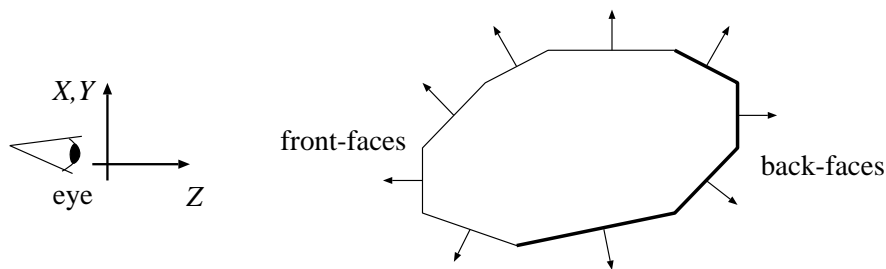
Figure 6.5: Normal vectors and back-faces

A usual agreement is, furthermore, that the normal vector at any object surface point (the normal vector of the tangent plane at that point) is defined so that it always points *outwards* from the object, as illustrated in figure 6.5. What can be stated about a surface point where the surface normal vector has a positive Z-coordinate (in the screen coordinate system)? It is definitely hidden from the eye since no light can depart from that point towards the eye! Roughly one half of the object surfaces is hidden because of this reason — and independently from other objects —, hence it is worth eliminating them from visibility calculations in advance. Object surfaces are usually decomposed into smaller parts called *faces*. If the normal vector at each point of a face has a positive Z-coordinate then it is called a **back-face** (see figure 6.5).

If a face is planar, then it has a unique normal vector, and the **back-face culling** (deciding whether it is a back-face) is not too expensive computationally. Defining one more convention, the vertices of planar polygonal faces can be numbered in counter-clockwise order, for example, looking from outside the object. If the vertices of this polygon appear in clockwise order on the image plane then the polygon is a back-face. How can it be detected? If $\vec{r}_1, \vec{r}_2, \vec{r}_3$ are three consecutive and non-collinear vertices of the polygon, then its normal vector, $\vec{n}$, can be calculated as:

$$\vec{n} = (-1)^c \cdot (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1) \tag{6.45}$$

where $c = 0$ if the inner angle at vertex $\vec{r}_2$ is less than π and $c = 1$ otherwise. If the Z-coordinate of $\vec{n}$ is positive, then the polygon is a back-face and can be discarded. If it is zero, then the projection of the polygon degenerates to a line segment and can also be discarded. A more tricky way of computing $\vec{n}$ is calculating the projected areas A_x, A_y, A_z of the polygon onto the planes perpendicular to the x-, y- and z-axes, respectively, and then taking $\vec{n}$ as the vector of components A_x, A_y, A_z. If the polygon vertices are given by the coordinates $(x_1, y_1, z_1), \ldots, (x_m, y_m, z_m)$ then the projected area A_z, for example, can be calculated as:

$$A_z = \frac{1}{2} \sum_{i=1}^{m} (x_{i \oplus 1} - x_i)(y_i + y_{i \oplus 1}) \tag{6.46}$$

where $i \oplus 1 = i + 1$ if $i < m$ and $m \oplus 1 = 1$. This method is not sensitive to collinear vertices and averages the errors coming from possible non-planarity of the polygon.

Note that if the object scene consists of nothing more than a single convex polyhedron, then the visibility problem can completely be solved by back-face culling: back-faces are discarded and non-back-faces are painted.

6.3   z-buffer algorithm

Another possible method for finding the visible object in individual pixels is that, for each object, all the pixels forming the image of the object on the screen are identified, and then, if a collision occurs at a given pixel due to overlapping, it is decided which object must be retained. The objects

are taken one by one. To generate all the pixels that the projection of an object covers, **scan conversion** methods can be used to convert the area of the projections first into (horizontal) spans corresponding to the rows of the raster image, and then split up the spans into pixels according to the columns. Imagine another array behind the raster image (raster buffer), with the same dimensions, but containing distance information instead of color values. This array is called **z-buffer**. Each pixel in the raster buffer has a corresponding cell in the z-buffer. This contains the distance (depth) information of the surface point from the eye which is used to decide which pixel is visible. Whenever a new color value is to be written into a pixel during the raster conversion of the objects, the value already in the z-buffer is compared with that of the actual surface point. If the value in the z-buffer is greater, then the pixel can be overwritten, both the corresponding color and depth information, because the actual surface point is closer to the eye. Otherwise the values are left untouched.

The basic form of the z-buffer algorithm is then:

> Initialize raster buffer to background color;
> Initialize each cell of zbuffer[] to ∞;
> **for** each object o **do**
>   **for** each pixel p covered by the projection of o **do**
>   **if** Z-coordinate of the surface point $<$ zbuffer[p] **then**
>   color of p = color of surface point;
>   zbuffer[p] = depth of surface point;
>   **endif**
>   **endfor**
> **endfor**

The value ∞ loaded into each cell of the z-buffer in the initialization step symbolizes the greatest possible Z value that can occur during the visibility calculations, and it is always a finite number in practice. This is also an image-precision algorithm, just like ray tracing. Its effectiveness can be — and usually is — increased by combining it with **back-face culling**.

The z-buffer algorithm is not expensive computationally. Each object is taken only once, and the number of operations performed on one object is proportional to the number of pixels it covers on the image plane. Having N objects $o_1, \ldots, o_N$, each covering P_i number of pixels individually on the

image plane, the time complexity T of the z-buffer algorithm is:

$$T = O \left(N + \sum_{i=1}^{N} P_i \right). \tag{6.47}$$

Since the z-buffer algorithm is usually preceded by a clipping operation discarding parts of objects outside the viewing volume, the number of pixels covered by the input objects $o_1, \ldots, o_N$ is $P_i = O(R^2)$ (R^2 is the resolution of the screen), and hence the time complexity of the z-buffer algorithm can also be written as:

$$T = O(R^2 N). \tag{6.48}$$

6.3.1   Hardware implementation of the z-buffer algorithm

Having approximated the surface by a polygon mesh, the surface is given by the set of mesh vertices, which should have been transformed to the screen coordinate system. Without loss of generality, we can assume that the polygon mesh consists of triangles only (this assumption has the important advantage that three points are always on a plane and the triangle formed by the points is convex). The visibility calculation of a surface is thus a series of visibility computations for screen coordinate system triangles, allowing us to consider only the problem of the scan conversion of a single triangle. Let the vertices of the triangle in screen coordinates be $\vec{r}_1 = [X_1, Y_1, Z_1]$, $\vec{r}_2 = [X_2, Y_2, Z_2]$ and $\vec{r}_3 = [X_3, Y_3, Z_3]$ respectively. The scan conversion algorithms should determine the X, Y pixel addresses and the corresponding Z coordinates of those pixels which belong to this triangle (figure 6.6). If the X, Y pixel addresses are already available, then the calculation of the corresponding Z coordinate can exploit the fact that the triangle is on a plane, thus the Z coordinate is some linear function of the X, Y coordinates. This linear function can be derived from the equation of the plane, using the notation $\vec{n}$ and $\vec{r}$ to represent the normal vector and the points of the plane respectively:

$$\vec{n} \cdot \vec{r} = \vec{n} \cdot \vec{r}_1 \quad \text{where} \quad \vec{n} = (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1). \tag{6.49}$$

Let us denote the constant $\vec{n} \cdot \vec{r}_1$ by C, and express the equation in scalar form, substituting the coordinates of the vertices ($\vec{r} = [X, Y, Z(X, Y)]$) and
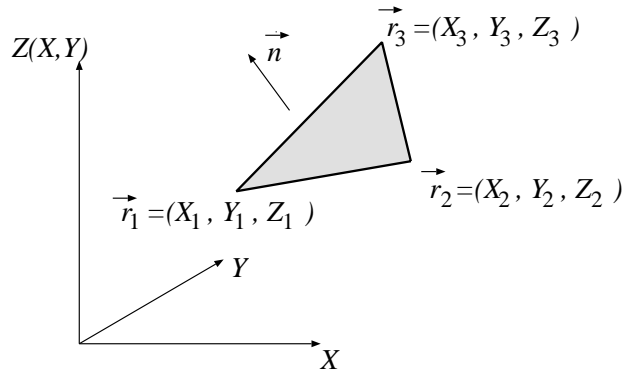
Figure 6.6: Screen space triangle

the normal of the plane ($\vec{n} = [n_X, n_Y, n_Z]$). The function of $Z(X, Y)$ is then:

$$Z(X, Y) = \frac{C - n_X \cdot X - n_Y \cdot Y}{n_Z}. \tag{6.50}$$

This linear function must be evaluated for those pixels which cover the pixel space triangle defined by the vertices $[X_1, Y_1]$, $[X_2, Y_2]$ and $[X_3, Y_3]$. Equation 6.50 is suitable for the application of the incremental concept discussed in subsection 2.3.2 on multi-variate functions. In order to make the boundary curve differentiable and simple to compute, the triangle is split into two parts by a horizontal line at the position of the vertex which is in between the other two vertices in the Y direction.

As can be seen in figure 6.7, two different orientations (called left and right orientations respectively) are possible, in addition to the different order of the vertices in the Y direction. Since the different cases require almost similar solutions, we shall discuss only the scan conversion of the lower part of a left oriented triangle, supposing that the Y order of the vertices is: $Y_1 < Y_2 < Y_3$.

The solution of the subsection 2.3.2 (on multi-variate functions) can readily be applied for the scan conversion of this part. The computational burden for the evaluation of the linear expression of the Z coordinate and for the calculation of the starting and ending coordinates of the horizontal spans of pixels covering the triangle can be significantly reduced by the incremental concept (figure 6.8).
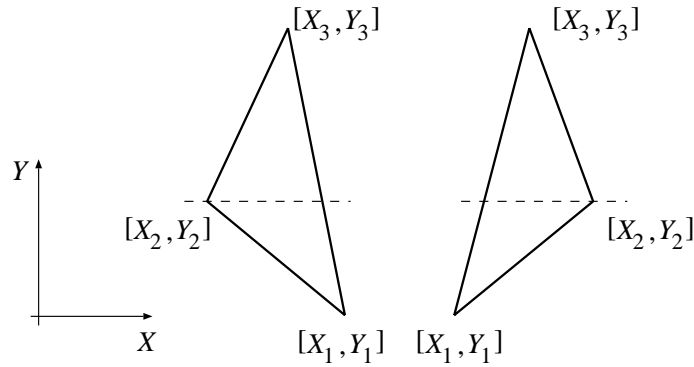
Figure 6.7: Breaking down the triangle

Expressing $Z(X+1, Y)$ as a function of $Z(X, Y)$, we get:

$$Z(X+1, Y) = Z(X, Y) + \frac{\partial Z(X, Y)}{\partial X} \cdot 1 = Z(X, Y) - \frac{n_X}{n_Z} = Z(X, Y) + \delta Z_X.$$
$$(6.51)$$

Since δZ_X does not depend on the actual X, Y coordinates, it has to be evaluated once for the polygon. In a scan-line, the calculation of a Z coordinate requires a single addition according to equation 6.51.

Since Z and X vary linearly along the left and right edges of the triangle, equations 2.33, 2.34 and 2.35 result in the following simple expressions in the range of $Y_1 \leq Y \leq Y_2$, denoting the K_s and K_e variables used in the general discussion by X_{start} and X_{end} respectively:

$$\begin{aligned}
X_{\text{start}}(Y+1) &= X_{\text{start}}(Y) + \frac{X_2 - X_1}{Y_2 - Y_1} = X_{\text{start}}(Y) + \delta X_Y^s \\
X_{\text{end}}(Y+1) &= X_{\text{end}}(Y) + \frac{X_3 - X_1}{Y_3 - Y_1} = X_{\text{end}}(Y) + \delta X_Y^e \\
Z_{\text{start}}(Y+1) &= Z_{\text{start}}(Y) + \frac{Z_2 - Z_1}{Y_2 - Y_1} = Z_{\text{start}}(Y) + \delta Z_Y^s \quad (6.52)
\end{aligned}$$
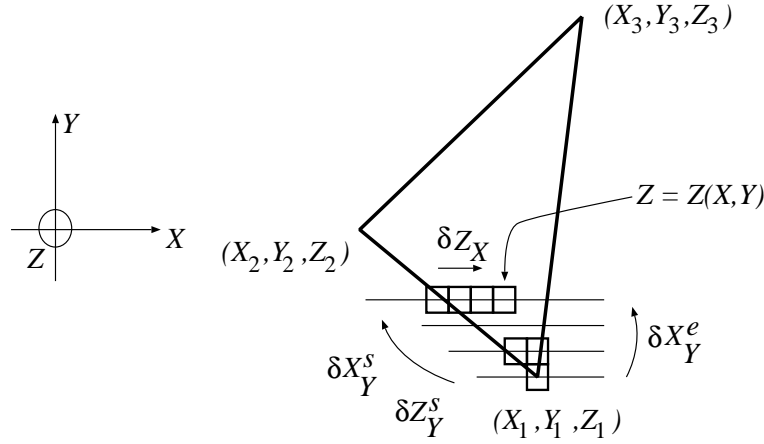
Figure 6.8: Incremental concept in Z-buffer calculations

The complete incremental algorithm is then:

$X_{\text{start}} = X_1 + 0.5;$ $X_{\text{end}} = X_1 + 0.5;$ $Z_{\text{start}} = Z_1 + 0.5;$
for $Y = Y_1$ **to** Y_2 **do**
    $Z = Z_{\text{start}};$
    for $X = \text{Trunc}(X_{\text{start}})$ **to** $\text{Trunc}(X_{\text{end}})$ **do**
        $z = \text{Trunc}(Z);$
        if $z < \text{Zbuffer}[X, Y]$ **then**
            raster_buffer$[X, Y]$ = computed color;
            $\text{Zbuffer}[X, Y] = z;$
        endif
        $Z\mathrel{+}= \delta Z_X;$
    endfor
    $X_{\text{start}} \mathrel{+}= \delta X_Y^s;$ $X_{\text{end}} \mathrel{+}= \delta X_Y^e;$ $Z_{\text{start}} \mathrel{+}= \delta Z_Y^s;$
endfor

Having represented the numbers in a fixed point format, the derivation of the executing hardware for this algorithm is straightforward following the methods outlined in section 2.3 on hardware realization of graphics algorithms.

6.4   Scan-line algorithm

The visibility problem can be solved separately for each horizontal row of
the image. This approach is a hybrid one, half way between image-precision
and object-precision methods. On the one hand, the so-called **scan-lines**
are discrete rows of the image, on the other hand, continuous calculations
are used at object-precision within the individual scan-lines. Such a hori-
zontal line corresponds to a horizontal plane in the screen coordinate system
(see left side of figure 6.9). For each such plane, we have to consider the
intersection of the objects with it. This gives two-dimensional objects on
the scan plane. If our object space consists of planar polygons, then a set of
line segments will appear on the plane. Those parts of these line segments
which are visible from the line $Z = 0$ have to be kept and drawn (see right
side of figure 6.9). If the endpoints of the segments are ordered by their X
coordinate, then the visibility problem is simply a matter of finding the line
segment with the minimal Z coordinate in each of the quadrilateral strips
between two consecutive X values. If the line segments can intersect, then
the X coordinates of the intersection points have also to be inserted into
the list of segment endpoints in order to get strips that are homogeneous
with respect to visibility, that is, with at most one segment visible in each.
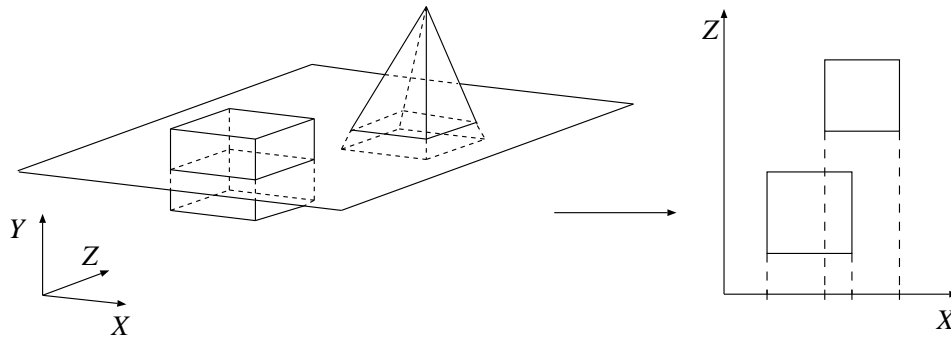
Figure 6.9: Scan-line algorithm

The basic outline of the algorithm is the following:

> **for** $Y = Y_{\min}$ **to** $Y_{\max}$ **do**
> > **for** each polygon P **do**
> > > compute intersection segments between P and plane at Y;
> >
> > **endfor**
> > sort endpoints of segments by their x coordinate;
> > compute and insert segment-segment intersection points;
> > **for** each strip s between two consecutive x values **do**
> > > find segment in s closest to axis x;
> > > draw segment;
> >
> > **endfor**
>
> **endfor**

If a given polygon intersects the horizontal plane at Y, it will probably intersect the next scan plane at $Y + 1$, as well. This is one of the guises of the phenomenon called **object coherence**. The origin of it is the basic fact that objects usually occupy compact and connected parts of space. Object coherence can be exploited in many ways in order to accelerate the calculations. In the case of the scan-line algorithm we can do the following. Before starting the calculation, we sort the maximal and minimal Y values of the polygons into a list called the **event list**. Another list, called the active polygon list, will contain only those polygons which really intersect the horizontal plane at the actual height Y. A Y coordinate on the event list corresponds either to the event of a new polygon being inserted into the active polygon list, or to the event of a polygon being deleted from it. These two lists will then be considered when going through the consecutive Y values in the outermost loop of the above algorithm. This idea can be refined by managing an active edge list (and the corresponding event list) instead of the active polygon list. A further acceleration can be the use of differential line generators for calculating the intersection point of a given segment with the plane at $Y + 1$ if the point at Y is known.

The time complexity of the algorithm in its "brute-force" form, as sketched above, is proportional to the number of rows in the picture on the one hand, and to the number of objects on the other hand. If the resolution of the screen is R^2, and the object scene consists of disjoint polygons having a

total of n edges, then:

$$T = O(R \cdot n). \tag{6.53}$$

If the proposed event list is used, and consecutive intersection points (the X values at $Y + 1$) are computed by differential line generators, then the time complexity is reduced:

$$T = O(n \log n + R \log n). \tag{6.54}$$

The $O(n \log n)$ term appears because of the sorting step before building the event list, the origin of the $O(R \log n)$ term is that the calculated intersection points must be inserted into an ordered list of length $O(n)$.

6.5 Area subdivision methods

If a pixel of the image corresponds to a given object, then its neighbors usually correspond to the same object, that is, visible parts of objects appear as connected territories on the screen. This is a consequence of object coherence and is called **image coherence**.
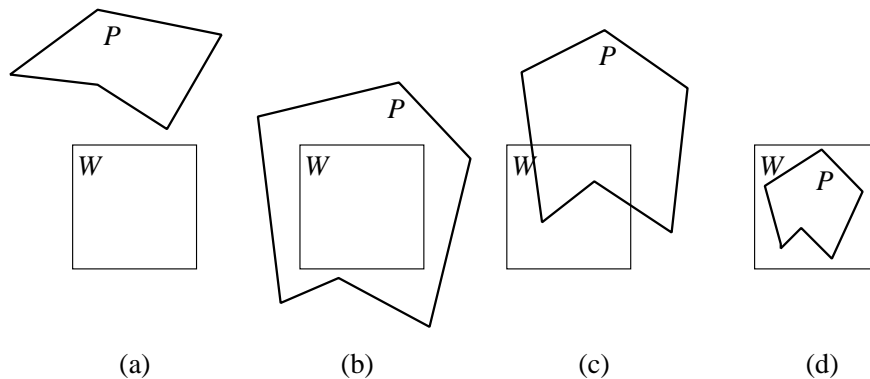
Figure 6.10: Polygon-window relations: distinct (a), surrounding (b), intersecting (c), contained (d)

If the situation is so fortunate — from a labor saving point of view — that a polygon in the object scene obscures all the others and its projection onto the image plane covers the image window completely, then we have to do no

more than simply fill the image with the color of the polygon. This is the basic idea of **Warnock's algorithm** [War69]. If no polygon edge falls into the window, then either there is no visible polygon, or some polygon covers it completely. The window is filled with the background color in the first case, and with the color of the closest polygon in the second case. If at least one polygon edge falls into the window, then the solution is not so simple. In this case, using a **divide-and-conquer** approach, the window is subdivided into four quarters, and each subwindow is searched recursively for a simple solution. The basic form of the algorithm rendering a rectangular window with screen (pixel) coordinates X_1, Y_1 (lower left corner) and X_2, Y_2 (upper right corner) is this:

> **Warnock**(X_1, Y_1, X_2, Y_2)
> > **if** $X_1 \neq X_2$ or $Y_1 \neq Y_2$ **then**
> > > **if** at least one edge falls into the window **then**
> > > > $X_m = (X_1 + X_2)/2;$
> > > > $Y_m = (Y_1 + Y_2)/2;$
> > > > **Warnock**$(X_1, Y_1, X_m, Y_m);$
> > > > **Warnock**$(X_1, Y_m, X_m, Y_2);$
> > > > **Warnock**$(X_m, Y_1, X_2, Y_m);$
> > > > **Warnock**$(X_m, Y_m, X_2, Y_2);$
> > > > **return** ;
> > > **endif**
> > **endif**
> > // *rectangle* X_1, Y_1, X_2, Y_2 *is homogeneous*
> > *polygon* = nearest to pixel $(X_1 + X_2)/2, (Y_1 + Y_2)/2;$
> > **if** no *polygon* **then**
> > > fill rectangle X_1, Y_1, X_2, Y_2 with background color;
> > **else**
> > > fill rectangle X_1, Y_1, X_2, Y_2 with color of *polygon*;
> > **endif**
> **end**

It falls into the category of image-precision algorithms. Note that it can handle non-intersecting polygons only. The algorithm can be accelerated by filtering out those polygons which can definitely not be seen in a given subwindow at a given step. Generally, a polygon can be in one of the fol-

lowing four kinds of relation with respect to the window, as shown in figure 6.10. A *distinct* polygon has no common part with the window; a *surrounding* polygon contains the window; at least one edge of an *intersecting polygon* intersects the border of the window; and a *contained* polygon falls completely within the window. Distinct polygons should be filtered out at each step of recurrence. Furthermore, if a surrounding polygon appears at a given stage, then all the others behind it can be discarded, that is all those which fall onto the opposite side of it from the eye. Finally, if there is only one contained or intersecting polygon, then the window does not have to be subdivided further, but the polygon (or rather the clipped part of it) is simply drawn. The price of saving further recurrence is the use of a scan-conversion algorithm to fill the polygon.

The time complexity of the Warnock algorithm is not easy to analyze, even for its initial form (sketched above). It is strongly affected by the actual arrangement of the polygons. It is easy to imagine a scene where each image pixel is intersected by at least one (projected) edge, from where the algorithm would go down to the pixel level at each recurrence. It gives a very poor worst-case characteristic to the algorithm, which is not worth demonstrating here. A better characterization would be an average-case analysis for some proper distribution of input polygons, which again length constraints of this book do not permit us to explore.

The Warnock algorithm recursively subdivides the screen into rectangular regions, irrespective of the actual shape of the polygons. It introduces superfluous vertical and horizontal edges. Weiler and Atherton [WA77] (also in [JGMHe88]) refined Warnock's idea from this point of view. The **Weiler–Atherton** algorithm also subdivides the image area recursively, but using the boundaries of the actual polygons instead of rectangles. The calculations begin with a rough **initial depth sort**. It puts the list of input polygons into a rough depth priority order, so that the "closest" polygons are in the beginning of the list, and the "farthest" ones at the end of it. At this step, any reasonable criterion for a sorting key is acceptable. The resulting order is not at all mandatory but increases the efficiency of the algorithm. Such a sorting criterion can be, for example, the smallest Z-value (Z_{min}) for each polygon (or Z_{max}, as used by the Newell–Newell–Sancha algorithm, see later). This sorting step is performed only once, at the beginning of the calculations, and is not repeated.

Let the resulting list of polygons be denoted by $L = \{P_1, \ldots, P_n\}$. Having done the sorting, the first polygon on the list (P_1) is selected. It is used to clip the remainder of the list into two new lists of polygons: the first list, say $I = \{P_1^I, \ldots, P_m^I\}$ ($m \leq n$), will contain those polygons — or parts of polygons — that fall *inside* the clip polygon P_1, and the second list, say $O = \{P_1^O, \ldots, P_M^O\}$ ($M \leq n$), will contain those ones that fall *outside* P_1. Then the algorithm examines the inside list I and removes all polygons located behind the current clip polygon since they are hidden from view. If the remaining list I' contains no polygon (the clip polygon obscures all of I), then the clip polygon is drawn and the initial list L is replaced by the outside list O and examined in a similar way to L. If the remaining list I' contains at least one polygon — that is, at least one polygon falls in front of the clip polygon — then it means that there was an error in the initial rough depth sort. In this case the (closest) offending polygon is selected as the clip polygon, and the same process is performed on list I' recursively, as on the initially ordered list L. Note that although the original polygons may be split into several pieces during the recursive subdivision, the clipping step (generating the lists I and O from L) can always be performed by using the original polygon corresponding to the actual clip polygon (which itself may be a clipped part of the original polygon). Maintaining a copy of each original polygon needs extra storage, but it reduces time.

There is, however, a more serious danger of clipping to the original copy of the polygons instead of their remainders! If there is *cyclic overlapping* between the original polygons, see figure 6.11 for example, then it can cause infinite recurrence of the algorithm. In order to avoid this, a set S of polygon names (references) is maintained during the process. Whenever a polygon P is selected as the clip polygon, its name (a reference to it) is inserted into S, and if it is processed (drawn or removed), its name is deleted from S. The insertion is done, however, only if P is not already in S, because if it is, then a cyclic overlap has been detected, and no additional recurrence is necessary because all polygons behind P have already been removed.

There is another crucial point of the algorithm: even if the scene consists only of convex polygons, the clipping step can quickly yield non-convex areas and holes (first when producing an outside list and then concavity is inherited by polygons in the later inside lists, as well). Thus, the polygon clipper has to be capable of clipping concave polygons with holes to both the inside and outside of a concave polygon with holes. Without going

into further details here, the interested reader is referred to the cited work [WA77], and only the above sketched ideas are summarized in the following pseudo-code:

WeilerAtherton(L)
    P = the first item on L;
    if $P \in S$ **then** draw P; **return** ; **endif**
    insert P into S;
    $I = \textbf{Clip}(L, P)$;
    $O = \textbf{Clip}(L, \overline{P})$;   // $\overline{P}$: complement of P
    for each polygon $Q \in I$;
        if Q is behind P **then**
           remove Q from I;
           if $Q \in S$ **then** remove Q from S; **endif**
        endif
    endfor
    if $I = \{\}$ **then**
        draw P;
        delete P from S;
    else
        WeilerAtherton(I);
    endif
    WeilerAtherton(O);
end

The recursive algorithm is called with the initially sorted list L of input polygons at the "top" level after initializing the set S to $\{\}$.

6.6   List-priority methods

Assume that the object space consists of planar polygons. If we simply scan convert them into pixels and draw the pixels onto the screen without any examination of distances from the eye, then each pixel will contain the color of the last polygon falling onto that pixel. If the polygons were ordered by their distance from the eye, and we took the farthest one first and the closest one last, then the final picture would be correct. Closer polygons would

obscure farther ones — just as if they were painted an opaque color. This (object-precision) method, is really known as the **painter's algorithm**.
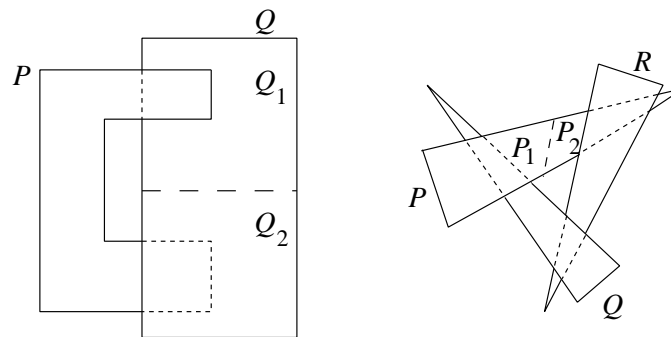
Figure 6.11: Examples for cyclic overlapping

The only problem is that the order of the polygons necessary for performing the painter's algorithm, the so-called **depth order** or **priority** relation between the polygons, is not always simple to compute.

We say that a polygon P *obscures* another polygon Q, if at least one point of Q is obscured by P. Let us define the relation $\succ$ between two polygons P and Q so that $P \succ Q$ if Q *does not obscure* P. If the relations $P_1 \succ P_2 \succ \ldots \succ P_n$ hold for a sequence of polygons, then this order coincides with the priority order required by the painter's algorithm. Indeed, if we drew the polygons by starting with the one furthest to the right (having the lowest priority) and finishing with the one furthest to the left, then the picture would be correct. However, we have to contend with the following problems with the relation $\succ$ defined this way:

1. If the projection of polygons P and Q do not overlap on the image plane, then $P \succ Q$ and $P \prec Q$, both at the same time, that is, the relation $\succ$ is *not antisymmetric*.

2. Many situations can be imagined, when $P \not\succ Q$ and $Q \not\succ P$ at the same time (see figure 6.11 for an example), that is, the relation $\succ$ is not defined for each pair of polygons.

3. Many situations can be imagined when a cycle $P \succ Q \succ R \succ P$ occurs (see figure 6.11 again), that is, the relation $\succ$ is *not transitive*.

The above facts prevent the relation $\succ$ from being an ordering relation, that is, the depth order is generally impossible to compute (at least if the polygons are not allowed to be cut). The first problem is not a real problem since polygons that do not overlap on the image plane can be painted in any order. What the second and third problems have in common is that both of them are caused by *cyclic overlapping* on the image plane. Cycles can be resolved by properly cutting some of the polygons, as shown by dashed lines in figure 6.11. Having cut the "problematic" polygons, the relation between resulting polygons will be cycle-free (transitive), that is $Q_2 \succ P \succ Q_1$ and $P_1 \succ Q \succ R \succ P_2$ respectively.
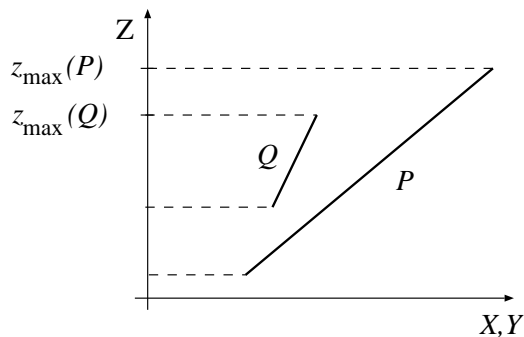
Figure 6.12: A situation when $z_{\max}(P) > z_{\max}(Q)$ *yet* $P \not\succ Q$

The **Newell–Newell–Sancha** algorithm [NNS72], [NS79] is one approach for exploiting the ideas sketched above. The first step is the calculation of an **initial depth order**. This is done by sorting the polygons according to their maximal z value, $z_{\max}$, into a list L. If there are no two polygons whose z ranges overlap, the resulting list will reflect the correct depth order. Otherwise, and this is the general case except for very special scenes such as those consisting of polygons all perpendicular to the z direction, the calculations need more care. Let us first take the polygon P which is the last item on the resulting list. If the z range of P does not overlap with any of the preceding polygons, then P is correctly positioned, and the polygon preceding P can be taken instead of P for a similar examination. Otherwise (and this is the general case) P overlaps a set $\{Q_1, \ldots, Q_m\}$ of polygons. This set can be found by scanning L from P backwards and taking the

consecutive polygons Q while $z_{\max}(Q) > z_{\min}(P)$. The next step is to try to check that P *does not* obscure any of the polygons in $\{Q_1, \ldots, Q_m\}$, that is, that P is at its right position despite the overlapping. A polygon P does not obscure another polygon Q, that is $Q \succ P$, if any of the following conditions holds:

1. $z_{\min}(P) > z_{\max}(Q)$ (they do not overlap in z range, this is the so-called z minimax check);

2. the bounding rectangle of P on the x, y plane does not overlap with that of Q (x, y minimax check);

3. each vertex of P is farther from the viewpoint than the plane containing Q;

4. each vertex of Q is closer to the viewpoint than the plane containing P;

5. the projections of P and Q do not overlap on the x, y plane.

The order of the conditions reflects the complexity of the check, hence it is worth following this order in practice. If it turns out that P obscures Q ($Q \not\succ P$) for a polygon in the set $\{Q_1, \ldots, Q_m\}$, then Q has to be moved behind P in L. This situation is illustrated in figure 6.12. Naturally, if P intersects Q, then one of them has to be cut into two parts by the plane of the other one. Cycles can also be resolved by cutting. In order to accomplish this, whenever a polygon is moved to another position in L, we mark it. If a marked polygon Q is about to be moved again because, say $Q \not\succ P$, then — assuming that Q is a part of a cycle — Q is cut into two pieces Q_1, Q_2, so that $Q_1 \not\succ P$ and $Q_2 \succ P$, and only Q_1 is moved behind P. A proper cutting plane is the plane of P, as illustrated in figure 6.11.

Considering the Newell–Newell–Sancha algorithm, the following observation is worth mentioning here. For any polygon P, let us examine the two halfspaces, say H_P^+ and H_P^-, determined by the plane containing P. If the viewing position is in H_P^+, then for all $p \in H_P^+$, P *cannot obscure* p, and for all $p \in H_P^-$, p cannot obscure P. On the other hand, if the viewing position is contained by H_P^-, similar observations can be made with the roles of H_P^+ and H_P^- interchanged. A complete algorithm for computing the depth order of a set $S = \{P_1, \ldots, P_n\}$ of polygons can be constructed based on this

idea, as proposed by Fuchs *et al.* [FKN80]. First P_i, one of the polygons, is selected. Then the following two sets are computed:

$$S_i^+ = (S \setminus P_i) \cap H_i^+, \quad S_i^- = (S \setminus P_i) \cap H_i^-, \quad (|S_i^+|, |S_i^-| \leq |S| - 1 = n - 1). \tag{6.55}$$

Note that some (if not all) polygons may be cut into two parts during the construction of the sets. If the viewing point is in H_i^+, then P_i cannot obscure any of the polygons in S_i^+, and no polygon in S_i^- can obstruct P_i. If the viewing point is in H_i^-, then the case is analogous with the roles of S_i^+ and S_i^- interchanged. That is, the position of P_i in the depth order is *between* those of the polygons in S_i^+ and S_i^-. The depth order in S_i^+ and S_i^- can then be recursively computed: a polygon P_j is selected from S_i^+ and the two sets S_j^+, S_j^- are created, and a polygon P_k is selected from S_i^- and the two sets S_k^+, S_k^- are created, etc. The subdivision is continued until the resulting set $S_{\cdot}^{\cdot}$ contains not more than one polygon (the depth order is then obvious in $S_{\cdot}^{\cdot}$; the dots in the subscript and superscript places stand for any possible value). This stop condition will definitely hold, since the size of both resultant sets $S_{\cdot}^+, S_{\cdot}^-$ is always at least one smaller than that of $S_{\cdot}^{\cdot}$, from which they are created (cf. equation 6.55).
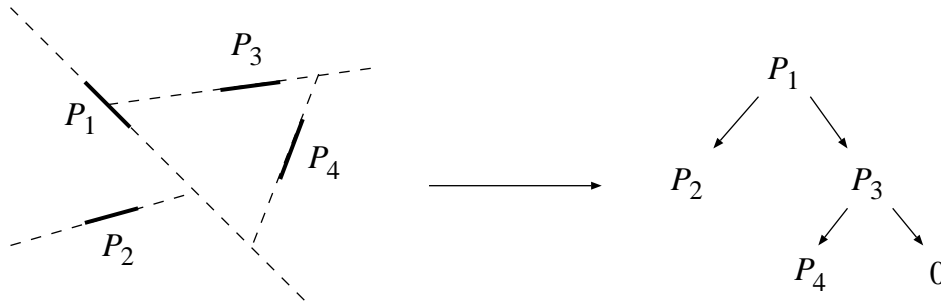
Figure 6.13: A binary space partitioning and its BSP-tree representation

The creation of the sets induces a subdivision of the object space, the so-called **binary space partitioning** (BSP) as illustrated in figure 6.13: the first plane divides the space into two halfspaces, the second plane divides the first halfspace, the third plane divides the second halfspace, further planes split the resulting volumes, etc. The subdivision can well be represented by a binary tree, the so-called **BSP-tree**, also illustrated in figure 6.13: the

first plane is associated with the root node, the second and third planes are associated with the two children of the root, etc. For our application, not so much the planes, but rather the polygons defining them, will be assigned to the nodes of the tree, and the set S of polygons contained by the volume is also necessarily associated with each node. Each leaf node will then contain either no polygon or one polygon in the associated set S (and no partitioning plane, since it has no child). The algorithm for creating the BSP-tree for a set S of polygons can be the following, where $S(N), P(N), L(N)$ and $R(N)$ denote the set of polygons, the "cutting" polygon and the left and right children respectively, associated with a node N:

BSPTree(S)
  create a new node N;
  $S(N) = S$;
  if $|S| \leq 1$ **then**
   $P(N) = null;\ L(N) = null;\ R(N) = null;$
  else
   $P = $ **Select**$(S);\ P(N) = P;$
   create sets S_P^+ and S_P^-;
   $L(N) = $ **BSPTree**$(S_P^+);$
   $R(N) = $ **BSPTree**$(S_P^-);$
  endif
  return N;
 end

The size of the BSP-tree, that is, the number of polygons stored in it, is on the one hand highly dependent on the nature of the object scene, and on the other hand on the "choice strategy" used by the routine **Select**. We can affect only the latter. The creators of the algorithm also proposed a heuristic choice criterion (without a formal proof) [FKN80], [JGMHe88] for minimizing the number of polygons in the BSP-tree. The strategy is two-fold: it minimizes the number of polygons that are split, and at the same time tries to maximize the number of "polygon conflicts" eliminated by the choice. Two polygons are in conflict if they are in the same set, and the plane of one polygon intersects the other polygon. What hoped for when maximizing the elimination of polygon conflicts is that the number of polygons which will need to be split in the descendent subtrees can be

reduced. In order to accomplish this, the following three sets are associated with each polygon P in the actual (to-be-split) set S:

$$
\begin{aligned}
S_1 &= \left\{ Q \in S \mid Q \text{ is entirely in } H_P^+ \right\}, \\
S_2 &= \left\{ Q \in S \mid Q \text{ is intersected by the plane of } P \right\}, \\
S_3 &= \left\{ Q \in S \mid Q \text{ is entirely in } H_P^- \right\}.
\end{aligned}
\tag{6.56}
$$

Furthermore, the following functions are defined:

$$
f(P,Q) = \begin{cases} 1, & \text{if the plane of } P \text{ intersects } Q; \\ 0, & \text{otherwise;} \end{cases}
\tag{6.57}
$$

$$
I_{i,j} = \sum_{P \in S_i} \sum_{Q \in S_j} f(P,Q)
$$

Then the routine **Select**(S) will return that polygon $P \in S$, for which the expression $I_{1,3} + I_{3,1} + w \cdot |S_2|$ is maximal, where w is a weight factor. The actual value of the weight factor w can be set based on practical experiments.

Note that the BSP-tree computed by the algorithm is *view-independent*, that is it contains the proper depth order for any viewing position. Differences caused by different viewing positions will appear in the manner of traversing the tree for retrieving the actual depth order. Following the characteristics of the BSP-tree, the traversal will always be an **inorder traversal**. Supposing that some action is to be performed on each node of a binary tree, the inorder traversal means that for each node, first one of its children is traversed (recursively), then the action is performed on the node, and finally the other child is traversed. This is in contrast to what happens with **preorder** or **postorder** traversals, where the action is performed before or after traversing the children respectively. The action for each node N here is the drawing of the polygon $P(N)$ associated with it. If the viewing position is in $H_{P(N)}^+$, then first the right subtree is drawn, then the polygon $P(N)$, and finally the left subtree, otherwise the order of the left and right children is back to front.

The following algorithm draws the polygons of a BSP-tree N in their proper depth order:

BSPDraw(N)
      if N is empty **then return** ;
      if the viewing position is in $H^+_{P(N)}$ **then**
         BSPDraw$(R(N))$; **Draw**$(P(N))$; **BSPDraw**$(L(N))$;
      else
         BSPDraw$(L(N))$; **Draw**$(P(N))$; **BSPDraw**$(R(N))$;
      endif
end

Once the BSP-tree has been created by the algorithm **BSPTree**, subsequent images for subsequent viewing positions can be generated by subsequent calls to the algorithm **BSPDraw**.

6.7   Planar graph based algorithms

A **graph** G is a pair $G(V, E)$ in its most general form, where V is the set of vertices or nodes, and E is the set of edges or arcs, each connecting two nodes. A graph is planar if it can be drawn onto the plane so that no two arcs cross each other. A **straight line planar graph (SLPG)** is a concrete embedding of a planar graph in the plane where all the arcs are mapped to (non-crossing) straight line segments. Provided that the graph is connected, the "empty" regions surrounded by an alternating chain of vertices and edges, and containing no more of them in the interior, are called faces. (Some aspects of these concepts were introduced briefly in section 1.6.2 on B-rep modeling.)

One of the characteristics of image coherence is that visible parts of objects appear as connected territories on the screen. If we have calculated these territories exactly, then we have only to paint each of them with the color of the corresponding object. Note that although the calculations are made on the image plane, this is an object-precision approach, because the accuracy of the result — at least in the first step — does not depend on the resolution of the final image. If the object scene consists of planar polygons, then the graph of visible parts will be a straight line planar graph,

also called the **visibility map** of the objects on the image plane. Its nodes and arcs correspond to the vertices and edges of polygons and intersections between polygons, and the faces represent homogeneous visible parts. We use the terms nodes and arcs of G in order to distinguish them from the vertices and edges of the polyhedra in the scene.

Let us assume in this section that the polygons of the scene do not intersect, except in cases when two or more of them share a common edge or vertex. This assumption makes the treatment easier, and it is still general enough, because scenes consisting of disjoint polyhedra fall into this category. The interested reader is recommended to study the very recent work of Mark de Berg [dB92], where the proposed algorithms can handle scenes of arbitrary (possibly intersecting) polygons. A consequence of our assumption is that the set of projected edges of the polygons is a superset of the set of edges contained in the visibility map. This is not so for the vertices, because a new vertex can occur on the image plane if a polygon partially obscures an edge. But the set of such new vertices is contained in the set of all intersection points between the projected edges. Thus we can first project all the polygon vertices and edges onto the image plane, then determine all the intersection points between the projected edges, and finally determine the parts that remain visible.
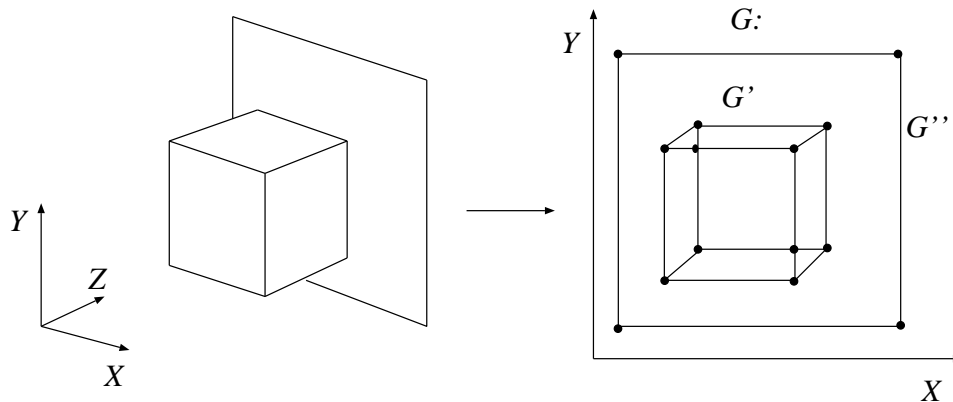
Figure 6.14: Example scene and the corresponding planar subdivision

In actual fact what we will do is to compute the graph G corresponding to the subdivision of the image plane induced by the projected vertices, edges

and the intersection between the edges. This graph will not be exactly the visibility map as we defined above, but will possess the property that the visibility will not change within the regions of the subdivision (that is the faces of the graph). Once we have computed the graph G, then all we have to do is visit its regions one by one, and for each region, we select the polygon closest to the image plane and use its color to paint the region. Thus the draft of the drawing algorithm for rendering a set $P_1, \ldots, P_N$ of polygons is the following:

1. project vertices and edges of $P_1, \ldots, P_N$ onto image plane;
2. calculate all intersection points between projected edges;
3. compute G, the graph of the induced planar subdivision;
4. **for** each region R of G **do**
5. P = the polygon visible in R;
6.   **for** each pixel p covered by R **do**
7.   color of p = color of P;
8.   **endfor**
9. **endfor**

The speed of the algorithm is considerably affected by how well its steps are implemented. In fact, all of them are critical, except for steps 1 and 7. A simplistic implementation of step 2, for example, would test each pair of edges for possible intersection. If the total number of edges is n, then the time complexity of this calculation would be $O(n^2)$. Having calculated the intersection points, the structure of the subdivision graph G has to be built, that is, incident nodes and arcs have to be assigned to each other somehow. The number of intersection points is $O(n^2)$, hence both the number of nodes and the number of arcs fall into this order. A simplistic implementation of step 3 would search for the possible incident arcs for each node, giving a time complexity of $O(n^4)$. This itself is inadmissible in practice, not to mention the possible time complexity of the further steps. (This was a simplistic analysis of simplistic approaches.)

We will take the steps of the visibility algorithm sketched above one by one, and also give a worst-case analysis of the complexity of the solution used. The approach and techniques used in the solutions are taken from [Dév93].

Representing straight line planar graphs

First of all, we have to devote some time to a consideration of what data structures can be used for representing a straight line planar graph, say $G(V, E)$. If the "topology" of the graph is known, then the location of the vertices determines unambiguously all other geometric characteristics of the graph. But if we intend to manipulate a graph quickly, then the matter of "topological" representation is crucial, and it may well be useful to include some geometric information too. Let us examine two examples where the different methods of representation allow different types of manipulations to be performed quickly.
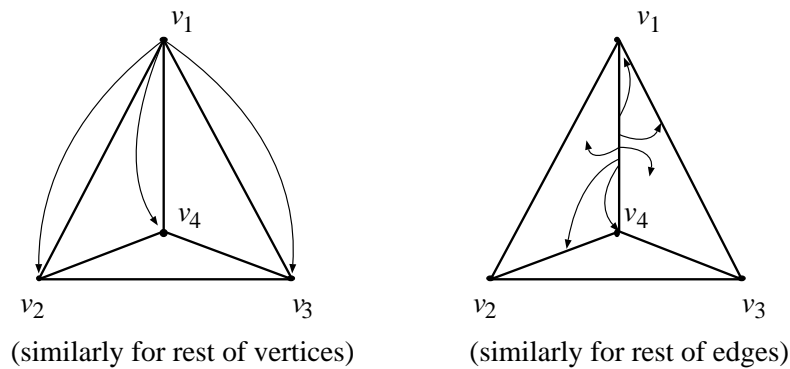
Figure 6.15: Adjacency lists and doubly connected edge list

The first scheme stores the structure by means of **adjacency lists**. Each vertex $v \in V$ has an adjacency list associated with it, which contains a reference to another vertex w, if there is an edge from v to w, that is $(v, w) \in E$. This is illustrated in figure 6.15. In the case of undirected graphs, each edge is stored twice, once at each of its endpoints. If we would like to "walk along" the boundary of a face easily (that is retrieve its boundary vertices and edges), for instance, then it is worth storing some extra information beyond that of the position of the vertices, namely the order of the adjacent vertices w around v. If adjacent vertices appear in counter clockwise order, for example, on the adjacency lists then walking around a face is easily achievable. Suppose that we start from a given vertex v of the face, and we know that the edge (v, w) is an edge of the face with

the face falling onto the right-hand side of it, where w is one of the vertices on the adjacency list of v. Then we search for the position of v on the adjacency list of w, and take the vertex next to v on this list as w', and w as v'. The edge (v', w') will be the edge next to (v, w) on the boundary of the face, still having the face on its right-hand side. Then we examine (v', w') in the same way as we did with (v, w), and step on, etc. We stop the walk once we reach our original (v, w) again. This walk would have been very complicated to perform without having stored the order of the adjacent vertices.

An alternative way of representing a straight line planar graph is the use of **doubly connected edge lists (DCELs)**, also shown in figure 6.15. The basic entity is now the edge. Each edge e has two vertex references, $v_1(e)$ and $v_2(e)$, to its endpoints, two edge references, $e_1(e)$ and $e_2(e)$, to the next edge (in counter clockwise order, for instance) around its two endpoints $v_1(e)$ and $v_2(e)$, and two face references, $f_1(e)$ and $f_2(e)$, to the faces sharing e. This type of representation is useful if the faces of the graph carry some specific information (for example: which polygon of the scene is visible in that region). It also makes it possible to traverse all the faces of the graph. The chain of boundary edges of a face can be easily retrieved from the edge references $e_1(e)$ and $e_2(e)$. This fact will be exploited by the following algorithm, which traverses the faces of a graph, and performs an action on each face f by calling a routine **Action**(f). It is assumed that each face has an associated mark field, which is initialized to *non-traversed*. The algorithm can be called with any edge e and one of its two neighboring faces f ($f = f_1(e)$ or $f = f_2(e)$).

> **Traverse**(e, f)
> > **if** f is marked as *traversed* **then return** ; **endif**
> > **Action**(f); mark f as *traversed*;
> > **for** each edge e' on the boundary of f **do**
> > > **if** $f_1(e') = f$ **then Traverse**$(e', f_2(e'))$;
> > > **else**   **Traverse**$(e', f_1(e'))$;
> > **endfor**
> **end**

Note that the algorithm can be used only if the faces of the graph contain no holes — that is the boundary edges of each face form a connected chain,

or, what is equivalent, the graph is connected. The running time T of the algorithm is proportional to the number of edges, that is $T = O(|E|)$, because each edge e is taken twice: once when we are on face $f_1(e)$ and again when we are on face $f_2(e)$.

If the graph has more than one connected component as the one shown in figure 6.14, then the treatment needs more care (faces can have holes, for example). In order to handle non-connected and connected graphs in a unified way, some modifications will be made on the DCEL structure. The unbounded part of the plane surrounding the graph will also be considered and represented by a face. Let this special face be called the *surrounding face*. Note that the surrounding face is always multiply connected (if the graph is non-empty), that is it contains at least one hole (in fact the edges of the hole border form the boundary edges of the graph), but has no boundary. We have already defined the structure of an edge of a DCEL structure, but no attention was paid to the structure of a face, although each edge has two explicit references to two faces. A face f will have a reference $e(f)$ to one of its boundary edges. The other boundary edges (except for those of the holes) can be retrieved by stepping through them using the DCEL structure. For the boundary of the holes, f will have the references $h_1(f), \ldots, h_m(f)$, where $m \geq 0$ is the number holes in f, each pointing to one boundary edge of the m different holes. Due to this modification, non-connected graphs will become connected from a computational point of view, and the algorithm **Traverse** will correctly visit all its faces, provided that the enumeration "**for** each edge e' on the boundary of f **do**" implies both the outer and the hole boundary edges. A proper call to visit each face of a possibly multiply connected graph is **Traverse**$(h_1(F), F)$, where F is the surrounding face.

Step 1: Projecting the edges

Let the object scene be a set of polyhedra, that is, where the faces of the objects are planar polygons. Assume furthermore that the boundary of the polyhedra (the structure of the vertices, edges and faces) is given by DCEL structures. (The DCEL structure used for boundary representation is known as the **winged edge** data structure for people familiar with shape modeling techniques.) This assumption is important because during the traversal of the computed visibility graph we will enter a new region by crossing one of its boundary edges, and we will have to know the polygon(s)

of the object scene the projection of which we leave or enter when crossing the edge on the image plane.

If the total number of edges is n, then the time T_1 required by this step is proportional to the number of edges, that is:

$$T_1 = O(n). \qquad (6.58)$$

Step 2: Calculating the intersection points

The second step is the calculation of the intersection points between the projected edges on the image plane. In the worst case the number of intersection points between n line segments can be as high as $O(n^2)$ (imagine, for instance, a grid of $n/2$ horizontal and $n/2$ vertical segments, where each of the horizontal ones intersects each of the vertical ones). In this worst case, therefore, calculation time cannot be better than $O(n^2)$, and an algorithm that compares each segment with all other ones would accomplish the task in optimal worst-case time. The running time of this algorithm would be $O(n^2)$, independently of the real number of intersections. We can create algorithms, however, the running time of which is "not too much" if there are "not too many" intersections. Here we give the draft of such an **output sensitive algorithm**, based on [Dév93] and [BO79]. Let us assume that no three line segments intersect at the same point and all the $2n$ endpoints of the n segments have distinct x-coordinates on the plane, a consequence of the latter being that no segments are vertical. Resolving these assumptions would cause an increase only in the length of the algorithm but not in its asymptotic complexity. See [BO79] for further details. Consider a vertical line $L(x)$ on the plane at a given abscissa x. $L(x)$ may or may not intersect some of our segments, depending on x. The segments $e_1, \ldots, e_k$ intersecting $L(x)$ at points $(x, y_1), \ldots, (x, y_k)$ appear in an ordered sequence if we walk along $L(x)$. A segment e_i is said to be *above* e_j at x if $y_i > y_j$. This relation is a total order for any set of segments intersecting a given vertical line. A necessary condition in order for two segments e_i and e_j to intersect is that there be some x at which e_i and e_j appear as neighbors in the order. All intersection points can be found by sweeping a vertical line in the horizontal direction on the plane and always comparing the neighbors in the order for intersection. The order along $L(x)$ can change when the abscissa x corresponds to one of the following: the left endpoint (beginning) of a segment,

the right endpoint (end) of a segment, and/or the intersection point of two segments. Thus our sweep can be implemented by stepping through only these specific positions, called *events*. The following algorithm is based on these ideas, which we can call as the **sweep-line** approach. It maintains a set Q for the event positions, a set R for the intersection points found and a set S for storing the order of segments along $L(x)$ at the actual position. All three sets are ordered, and for set S, succ(s) and prec(s) denote the successor and the predecessor of $s \in S$, respectively.

> Q = the set of all the $2n$ segment endpoints;
> $R = \{\}$; $S = \{\}$;
> sort Q by increasing x-values;
> **for** each point $p \in Q$ in increasing x-order **do**
>   **if** p is the left endpoint of a segment s **then**
>   insert s into S;
>   **if** s intersects succ(s) at any point q **then** insert q into Q;
>   **if** s intersects prec(s) at any point q **then** insert q into Q;
>   **else if** p is the right endpoint of a segment s **then**
>   **if** succ(s) and prec(s) intersect at any point q **then**
>   **if** $q \notin Q$ **then** insert q into Q;
>   **endif**
>   delete s from S
>   **else**   // *p is the intersection of segments s and t, say*
>   add p to R; swap s and t in S;   //*say s is above t*
>   **if** s intersects succ(s) at any point q **then**
>   **if** $q \notin Q$ **then** insert q into Q;
>   **endif**
>   **if** t intersects prec(t) at any q **then**
>   **if** $q \notin Q$ **then** insert q into Q;
>   **endif**
>   **endif**
> **endfor**

Note that the examinations "**if** $q \notin Q$" are really necessary, because the intersection of two segments can be found to occur many times (the appearance and disappearance of another segment between two segments can even occur $n - 2$ times!). The first three steps can be performed in $O(n \log n)$

time because of sorting. The main loop is executed exactly $2n + k$ times, where k is the number of intersection points found. The time complexity of one cycle depends on how sophisticated the data structures used for implementing the sets Q and S are, because insertions and deletions have to be performed on them. R is not crucial, a simple array will do. Since the elements of both Q and S have to be in order, an optimal solution is the use of balanced binary trees. Insertions, deletions and searching can be performed in $O(\log N)$ time on a balanced tree storing N elements (see [Knu73], for instance). Now $N = O(n^2)$ for Q and $N = O(n)$ for S, hence $\log N = O(\log n)$ for both. We can conclude that the time complexity of our algorithm for finding the intersection of n line segments in the plane, that is the time T_2 required by step 2 of the visibility algorithm is:

$$T_2 = O((n + k) \log n). \tag{6.59}$$

Such an algorithm is called an **output sensitive** algorithm, because its complexity depends on the actual size of the output. It is generally worth mentioning that if we have a problem with a very bad worst-case complexity due to the possible size of the output, although the usual size of the output is far less, then we have to examine whether an output sensitive algorithm can be constructed.

Step 3: Constructing the subdivision graph G

In step 3 of the proposed visibility algorithm we have to produce the subdivision graph G so that its faces can be traversed efficiently in step 4. A proper representation of G, as we have seen earlier, is a DCEL structure. It will be computed in two steps, first producing an intermediate structure which is then easily converted to a DCEL representation. We can assume that the calculations in steps 1 and 2 have been performed so that all the points — that is the projections of the $2n$ vertices and the k intersection points — have references to the edge(s) they lie on. First of all, for each edge we sort the intersection points lying on it (sorting is done along each edge, individually). Since $O(N \log N)$ time is sufficient (and also necessary) for sorting N numbers, the time consumed by the sorting along an edge e_i is $O(N_i \log N_i)$, where N_i is the number of intersection points to be sorted on e_i. Following from the general relation that if $N_1 + \ldots + N_n = N$, then

$$N_1 \log N_1 + \ldots + N_n \log N_n \leq N_1 \log N + \ldots + N_n \log N = N \log N, \tag{6.60}$$

the sum of the sorting time at the edges is $O(k \log k) = O(k \log n)$, since $N = 2k = O(n^2)$ (one intersection point appears on two segments). Having sorted the points along the edges, we divide the segments into subsegments at the intersection points. Practically speaking this means that the representation of each edge will be transformed into a doubly linked list, illustrated in figure 6.16. Such a list begins with a record describing its starting point.
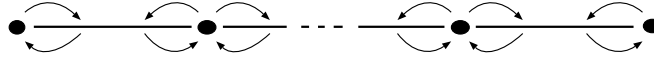
Figure 6.16: Representation of a subdivided segment

It is (doubly) linked to a record describing the first subsegment, which is further linked to its other endpoint, etc. The last element of the list stores the end point of the edge. The total time needed for this computation is $O(n + k)$, since there are $n + 2k$ subsegments. Note that each intersection point is duplicated although this could be avoided by modifying the representation a little. Note furthermore that if the real spatial edges corresponding to the projected edges $e_{i_1}, \ldots, e_{i_m}$ meet at a common vertex on the boundary of a polyhedron, then the projection of this common vertex is represented m times in our present structure. So we merge the different occurrences of each vertex into one. This can be done by first sorting the vertices in *lexicographic order* with respect to their x, y coordinates and then merging equal ones. Lexicographic ordering means that a vertex with coordinates x_1, y_1 precedes another one with coordinates x_2, y_2, if $x_1 < x_2$ or $x_1 = x_2 \wedge y_1 < y_2$. They are equal if $x_1 = x_2 \wedge y_1 = y_2$. The merging operation can be performed in $O((n+k) \log(n+k)) = O((n+k) \log n)$ time because of the sorting step. Having done this, we have a data structure for the subdivision graph G, which is similar to an adjacency list representation with the difference that there are not only vertices but edges too, and the neighbors (edges, vertices) are not ordered around the vertices. Ordering adjacent edges around the vertices can be done separately for each vertex. For a vertex v_i with N_i edges around it, this can be done in $O(N_i \log N_i)$ time. The total time required by the m vertices will be $O((n + k) \log n)$, using relation 6.60 again with $N_1 + \ldots + N_m = n + 2k$. The data structure obtained in this way is halfway between the adjacency list and the DCEL

representation of G. It is "almost" DCEL, since edges appear explicitly, and each edge has references to its endpoints. The two reasons for incompleteness are that no explicit representation of faces appears, and the edges have no explicit reference to the edges next to them around the endpoints — the references exist, however, but only implicitly through the vertices. Since the edges are already ordered about the vertices, these references can be made explicit by scanning all the edges around each vertex, which requires $O(n + k)$ time. The faces can be constructed by first generating the faces of the connected components of G separately, and then merging the DCEL structure of the components into one DCEL structure. The first step can be realized by using an algorithm very similar to **Traverse**, since the outer boundary of each face can be easily retrieved from our structure, because edges are ordered around vertices. Assuming that the face references $f_1(e), f_2(e)$ of each edge e are initialized to *null*, the following algorithm constructs the faces of G and links them into the DCEL structure:

> **for** each edge e **do MakeFaces**(e); **endfor**
> **MakeFaces**(e)
>   **for** $i = 1$ **to** 2 **do**
>   **if** $f_i(e) = null$ **then**
>   construct a new face f;
> $e(f) = e$; set m (the number of holes in f) to 0;
>   **for** each edge e' on the boundary of f **do**
>   **if** $f_1(e')$ corresponds to the side of f **then**
> $f_1(e') = f$;
>   **else**
> $f_2(e') = f$;
>   **endif**
>   **MakeFaces**(e');
>   **endfor**
>   **endif**
>   **endfor**
> **end**

Note that the recursive subroutine **MakeFaces**(e) traverses that connected component of G which contains the argument edge e. The time complexity of the algorithm is proportional to the number of edges, that

is $O(n + k)$, because each edge is taken at most three times (once in the main loop and twice when traversing the connected component containing the edge).

The resulting structure generally consists of more than one DCEL structure corresponding to the connected components of G. Note furthermore that the surrounding faces contain no holes. Another observation is that for any connected component G' of G the following two cases are possible: (1) G' falls onto the territory of at least one component (as in figure 6.14) and then it is contained by at least one face. (2) G' falls outside any other components (it falls into their surrounding face). In case (1) the faces containing G' form a nested sequence. Let the smallest one be denoted by f. Then for each boundary edge of G', the reference to the surrounding face of G' has to be substituted by a reference to f. Moreover, the boundary edges of G' will form the boundary of a hole in the face f, hence a new hole edge reference $h_{m+1}(f)$ (assuming that f has had m holes so far) has to be created for f, and $h_{m+1}(f)$ is to be set to one of the boundary edges of G'. In case (2) the situation is very similar, the only difference being that the surrounding face F corresponding to the resulting graph G plays the role of f. Thus the problem is first creating F, the "united" surrounding face of G, and then locating and linking the connected components of G in its faces. In order to accomplish this task efficiently, a **sweep-line** approach will be used.
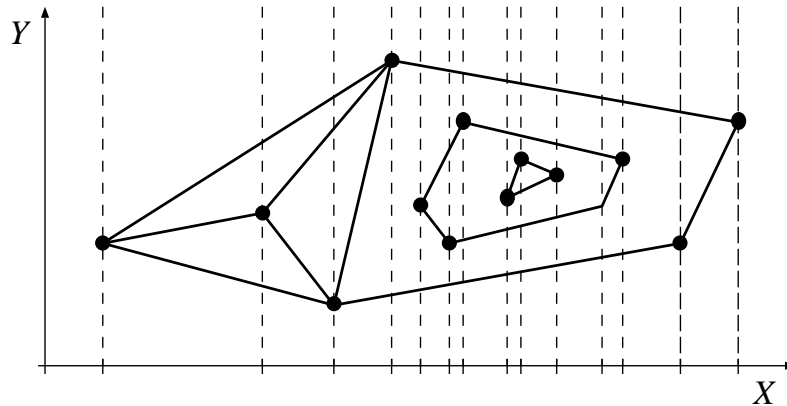
Figure 6.17: Slabs

The problem of locating a component, that is finding the face containing it, is equivalent to the problem of locating one of its vertices, that is, our problem is a point location problem. Imagine a set of vertical lines through each vertex of the graph G, as shown in figure 6.17. These parallel lines divide the plane into unbounded territories, called *slabs*. The number of slabs is $O(n + k)$. Each slab is divided into $O(n + k)$ parts by the crossing edges, and the crossing edges always have the same order along any vertical line in the interior of the slab. Given efficient data structures (with $O(\log(n+k))$ search time) for storing the slabs and for the subdivision inside the slabs, the problem of locating a point can be performed efficiently (in $O(\log(n + k))$ time). This is the basic idea behind the following algorithm which first determines the order of slabs, and then scans the slabs in order (from left to right) and incrementally constructs the data structure storing the subdivision. This data structure is a balanced binary tree, which allows efficient insertion and deletion operations on it. In order that the algorithm may be understood, two more notions must be defined. Each vertical line (beginning of a slab) corresponds to a vertex. The edges incident to this vertex are divided into two parts: the edges on the left side of the line are called *incoming edges*, while those on the right side are *outgoing edges*. If we imagine a vertical line sweeping the plane from left to right, then the names are quite apt. The vertex which is first encountered during the sweep — that is, the vertex furthest to the left — definitely corresponds to the boundary of the (first) hole of the surrounding face F, hence F can be constructed at this stage. (Note that this is so because the line segments are assumed to be straight.) Generally, if a vertex v with no incoming edges is encountered during the sweep (this is the case for the furthest left vertex too), it always denotes the appearance of a new connected component, which then has to be linked into the structure. The structure storing the subdivision of the actual slab (that is the edges crossing the actual slab) will be a balanced tree T.

The algorithm is the following:

> sort all the vertices of G by their (increasing) x coordinates;
> create F (the surrounding face);
> $T = \{\}$;
> **for** each vertex v in increasing x-order **do**
> > **if** v has only outgoing edges (a new component appears) **then**
> > > $f =$ the face containing v (search in T);
> > > mutually link f and the boundary chain containing v;
> > **endif**
> > **for** all the incoming edges e_{in} at v **do**
> > > delete e_{in} from T;
> > **endfor**
> > **for** all the outgoing edges e_{out} at v **do**
> > > insert e_{out} into T;
> > **endfor**
> **endfor**

The face f containing a given vertex v can be found by first searching for the place where v could be inserted into T, and then f can be retrieved from the edge either above or below the position of v in T. If T is empty, then $f = F$.

The sorting (first) step can be done in $O((n + k) \log(n + k)) = O((n + k) \log n)$ time; the main cycle is executed $O(n + k)$ times; the insertions into and deletions from T need only $O(\log(n + k)) = O(\log n)$ time. The time required to link the boundary of a connected component into the face containing it is proportional to the number of edges in the boundary chain, but each component is linked only once (when encountering its leftmost vertex), hence the total time required by linking is $O(n + k)$. Thus the running time of the algorithm is $O((n + k) \log n)$.

We have come up with a DCEL representation of the subdivision graph G, and we can conclude that the time T_3 consumed by step 3 of the visibility algorithm is:

$$T_3 = O((n + k) \log n). \tag{6.61}$$

Steps 4–9: Traversing the subdivision graph G

Note that it causes no extra difficulties in steps 1–3 to maintain two more references $F_1(e), F_2(e)$ for each edge e, pointing to the spatial faces incident to the original edge from which e has been projected (these are boundary faces of polyhedra in the object scene).

Steps 4–9 of the algorithm will be examined together. The problem is to visit each face of G, retrieve the spatial polygon closest to the image plane for the face, and then draw it. We have already proposed the algorithm **Traverse** for visiting the faces of a DCEL structure. Its time complexity is linearly proportional to the number of edges in the graph, if the action performed on the faces takes only a constant amount of time. We will modify this algorithm a little bit and examine the time complexity of the action. The basic idea is the following: for each face f of G, there are some spatial polygons, the projection of which completely covers f. Let us call them *candidates*. The projections of all the other polygons have empty intersection with f, hence they cannot be visible in f. Candidate polygons are always in a unique order with respect to their distance from the image plane (that is from f). The candidate polygon must always be retrieved at the first position. The candidate-set changes if we cross an edge of G. If we cross some edge e, then for each of the two spatial faces $F_1(e)$ and $F_2(e)$ pointed to by e there are two possibilities: either it appears as a new member in the set of candidates or it disappears from it, depending on which direction we cross e. Thus we need a data structure which is capable of storing the actual candidates in order, on which insertions and deletions can be performed efficiently, and where retrieving the first element can be done as fast as possible. The balanced binary tree would be a very good candidate were there not a better one: the **heap**. An N-element heap is a 1-dimensional array $H[1, \ldots, N]$, possessing the property:

$$H[i] \leq H[2i] \quad \text{and} \quad H[i] \leq H[2i + 1]. \tag{6.62}$$

Insertions and deletions can be done in $O(\log N)$ time [Knu73], just as for balanced binary trees, but retrieving the first element (which is always $H[1]$) requires only constant time. Initializing a heap H for storing the candidate polygons at any face f can be done in $O(n \log n)$ time, since $N = O(n)$ in our case (from Euler's law concerning the number of faces, edges and vertices of polyhedra). This has to be done only once before the traversal, because

H can be updated during the traversal when crossing the edges. Hence the time required for retrieving the closest polygon to any of the faces (except for the first one) will not be more than $O(\log n)$. The final step is the drawing (filling the interior) of the face with the color of the corresponding polygon. Basic 2D scan conversion algorithms can be used for this task. An arbitrary face f_i with N_i edges can be raster converted in $O(N_i \log N_i + P_i)$ time, where P_i is the number of pixels it covers (see [NS79]). The total time spent on raster converting the faces of G is $O((n + k) \log n + R^2)$, since $N_1 + \ldots + N_m = 2(n + 2k)$, and $P_1 + \ldots + P_m \leq R^2$ (no pixel is drawn twice), where R^2 is the resolution (number of pixels) of the screen. Thus the time T_4 required by steps 4–9 of the visibility algorithm is:

$$T_4 = O((n + k) \log n + R^2). \tag{6.63}$$

This four-step analysis shows that the time complexity of the proposed visibility algorithm, which first computes the visibility map induced by a set of non-intersecting polyhedra having n edges altogether, and then traverses its faces and fills them with the proper color, is:

$$T_1 + T_2 + T_3 + T_4 = O((n + k) \log n + R^2), \tag{6.64}$$

where k is the number of intersections between the projected edges on the image plane. It is not really an output sensitive algorithm, since many of the k intersection points may be hidden in the final image, but it can be called an *intersection sensitive* algorithm.

Chapter 7

INCREMENTAL SHADING TECHNIQUES

Incremental shading models take a very drastic approach to simplifying the rendering equation, namely eliminating all the factors which can cause multiple interdependence of the radiant intensities of different surfaces. To achieve this, they allow only coherent transmission (where the refraction index is 1), and incoherent reflection of the light from abstract lightsources, while ignoring the coherent and incoherent reflection of the light coming from other surfaces.

The reflection of the light from abstract lightsources can be evaluated without the intensity of other surfaces being known, so the dependence between them has been eliminated. In fact, coherent transmission is the only feature left which can introduce dependence, but only in one way, since only those objects can alter the image of a given object which are behind it, looking at the scene from the camera.

Suppose there are n_l abstract lightsources (either directional, positional or flood type) and that ambient light is also present in the virtual world. Since the flux of the abstract lightsources incident to a surface point can be easily calculated, simplifying the integrals to sums, the shading equation has the following form:

$$I^{\text{out}} = I_e + k_a \cdot I_a + k_t \cdot I_t + \sum_{l}^{n_l} r_l \cdot I_l \cdot k_d \cdot \cos \phi_{\text{in}} + \sum_{l}^{n_l} r_l \cdot I_l \cdot k_s \cdot \cos^n \psi \quad (7.1)$$

where k_a is the reflection coefficient of the ambient light, k_t, k_d and k_s are the

transmission, diffuse and specular coefficients respectively, ϕ_{in} is the angle between the direction of the lightsource and the surface normal, ψ is the angle between the viewing vector and the mirror direction of the incident light beam, n is the specular exponent, I_l and I_a are the incident intensities of the normal and ambient lightsources at the given point, I_t is the intensity of the surface behind a transmissive object, I_e is the own emission, and r_l is the shadow factor representing whether a lightsource can radiate light onto the given point, or whether the energy of the beam is attenuated by transparent objects, or whether the point is in shadow, because another opaque object is hiding it from the lightsource:

$$
r_l = \begin{cases}
1 \text{ if the lightsource } l \text{ is visible from this point} \\
\prod_i k_t^{(i)} \text{ if the lightsource is masked by transparent objects} \\
0 \text{ if the lightsource is hidden by an opaque object}
\end{cases}
\tag{7.2}
$$

where $k_t^{(1)}, k_t^{(2)}, \ldots, k_t^{(n)}$ are the transmission coefficients of the transparent objects between the surface point and lightsource l.

The factor r_l is primarily responsible for the generation of shadows on the image.

7.1 Shadow calculation

The determination of r_l is basically a visibility problem considering whether a lightsource is visible from the given surface point, or equally, whether the surface point is visible from the lightsource. Additionally, if there are transparent objects, the solution also has to determine the objects lying in the path of the beam from the lightsource to the surface point.

The second, more general case can be solved by ray-tracing, generating a ray from the surface point to the lightsource and calculating the intersections, if any, with other objects. In a simplified solution, however, where transparency is ignored in shadow calculations, that is where r_l can be either 0 or 1, theoretically any other visible surface algorithm can be applied setting the eye position to the lightsource, then determining the surface parts visible from there, and declaring the rest to be in shadow. The main difficulty of shadow algorithms is that they have to store the information

regarding which surface parts are in shadow until the shading calculation, or else that question has to be answered during shading of each surface point visible in a given pixel, preventing the use of coherence techniques and therefore limiting the possible visibility calculation alternatives to expensive ray-tracing.

An attractive alternative algorithm is based on the application of the z-buffer method, requiring additional z-buffers, so-called **shadow maps**, one for each lightsource (figure 7.1).
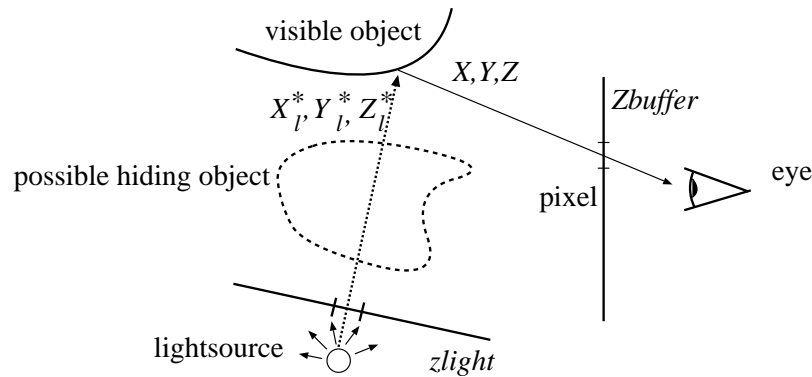
Figure 7.1: Shadow map method

The algorithm consists of a z-buffer step from each lightsource l setting the eye position to it and filling its shadow map $zlight_l[X, Y]$, then a single modified z-buffer step for the observer's eye position filling $Zbuffer[X, Y]$.

From the observer's eye position, having checked the visibility of the surface in the given pixel by the $Z_{\mathrm{surface}}[X, Y] < Zbuffer[X, Y]$ inequality, the algorithm transforms the 3D point $(X, Y, Z_{\mathrm{surface}}[X, Y])$ from the observer's eye coordinate system (screen coordinate system) to each lightsource coordinate system, resulting in:

$$(X, Y, Z_{\mathrm{surface}}[X, Y]) \overset{T}{\Longrightarrow} (X_l^*, Y_l^*, Z_l^*). \tag{7.3}$$

If $Z_l^* > zlight[X_l^*, Y_l^*]$, then the surface point was not visible from the lightsource l, hence, with respect to this lightsource, it is in shadow ($r_l = 0$).

The calculation of shadows seems time consuming, as indeed it is. In many applications, especially in CAD, shadows are not vital, and they can

even confuse the observer, making it possible to speed up image generation
by ignoring the shadows and assuming that $r_l = 1$.

7.2   Transparency

If there are no transparent objects image generation is quite straightforward
for incremental shading models. By applying a hidden-surface algorithm,
the surface visible in a pixel is determined, then the simplified shading
equation is used to calculate the intensity of that surface, defining the color
or (R, G, B) values of the pixel.

Should transparent objects exist, the surfaces have to be ordered in de-
creasing distance from the eye, and the shading equations have to be eval-
uated according to that order. Suppose the color of a "front" surface is
being calculated, when the intensity of the "back" surface next to it is al-
ready available (I_{back}), as is the intensity of the front surface, taking only
reflections into account (I_{front}^{ref}). The overall intensity of the front surface,
containing both the reflective and transmissive components, is:

$$I_{front}[X, Y] = I_{front}^{ref} + k_t \cdot I_{back}[X, Y]. \qquad (7.4)$$

The transmission coefficient, k_t, and the reflection coefficients are obvi-
ously not independent. If, for example, k_t were 1, all the reflection param-
eters should be 0. One way of eliminating that dependence is to introduce
corrected reflection coefficients by dividing them by $(1 - k_t)$, and calculating
the reflection I_{front}^* with these corrected parameters. The overall intensity
is then:

$$I_{front}[X, Y] = (1 - k_t) \cdot I_{front}^* + k_t \cdot I_{back}[X, Y]. \qquad (7.5)$$

This formula can be supported by a pixel level trick. The surfaces can be
rendered independently in order of their distance from the eye, and their
images written into the frame buffer, making a weighted sum of the reflective
surface color, and the color value already stored in the frame buffer (see also
subsection 8.5.3 on support for translucency and dithering).

7.3 Application of the incremental concept in shading

So far, the simplified shading equation has been assumed to have been evaluated for each pixel and for the surface visible in this pixel, necessitating the determination of the surface normals to calculate the angles in the shading equation.

The speed of the shading could be significantly increased if it were possible to carry out the expensive computation just for a few points or pixels, and the rest could be approximated from these representative points by much simpler expressions. These techniques are based on linear (or in extreme case constant) approximation requiring a value and the derivatives of the function to be approximated, which leads to the incremental concept. These methods are efficient if the geometric properties can also be determined in a similar way, connecting incremental shading to the incremental visibility calculations of polygon mesh models. Only polygon mesh models are considered in this chapter, and should the geometry be given in a different form, it has to be approximated by a polygon mesh before the algorithms can be used. It is assumed that the geometry will be transformed to the screen coordinate system suitable for visibility calculations and projection.

There are three accepted degrees of approximation used in this problem:

1. **Constant shading** where the color of a polygon is approximated by a constant value, requiring the evaluation of the shading equation once for each polygon.

2. **Gouraud shading** where the color of a polygon is approximated by a linear function, requiring the evaluation of the shading equation at the vertices of the polygon. The color of the inner points is determined by incremental techniques suitable for linear approximation.

3. **Phong shading** where the normal vector of the surface is approximated by a linear function, requiring the calculation of the surface normal at the vertices of the polygon, and the evaluation of the shading equation for each pixel. Since the color of the pixels is a non-linear function of the surface normal, Phong shading is, in fact, a non-linear approximation of color.
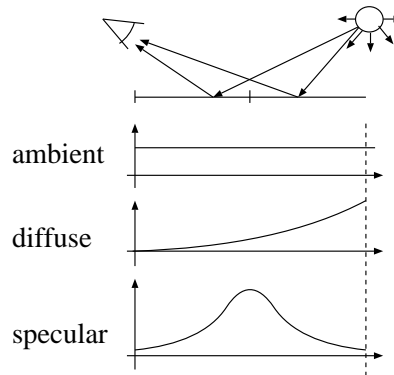
Figure 7.2: Typical functions of ambient, diffuse and specular components

In figure 7.2 the intensity distribution of a surface lit by positional and ambient lightsources is described in terms of ambient, diffuse and specular reflection components. It can be seen that ambient and diffuse components can be fairly well approximated by linear functions, but the specular term tends to show strong non-linearity if a highlight is detected on the surface. That means that constant shading is acceptable if the ambient lightsource is dominant, and Gouraud shading is satisfactory if k_s is negligible compared with k_d and k_a, or if there are no highlights on the surface due to the relative arrangement of the lightsources, the eye and the surface. If these conditions do not apply, then only Phong shading will be able to provide acceptable image free from artifacts.

Other features, such as shadow calculation, texture or bump mapping (see chapter 12), also introduce strong non-linearity of the intensity distribution over the surface, requiring the use of Phong shading to render the image.

7.4   Constant shading

When applying constant shading, the simplified rendering equation missing out the factors causing strong non-linearity is evaluated once for each polygon:

$$I^{\text{out}} = I_e + k_a \cdot I_a + \sum_{l}^{n_l} I_l \cdot k_d \cdot \max\{(\vec{N} \cdot \vec{L}), 0\}. \tag{7.6}$$

In order to generate the unit surface normal $\vec{N}$ for the formula, two alternatives are available. It can either be the "average" normal of the real surface over this polygon estimated from the normals of the real surface in the vertices of the polygon, or else the normal of the approximating polygon.

7.5   Gouraud shading

Having approximated the surface by a polygon mesh, Gouraud shading requires the evaluation of the rendering equation at the vertices for polygons, using the normals of the real surface in the formula. For the sake of simplicity, let us assume that the polygon mesh consists of triangles only (this assumption has an important advantage in that three points are always on a plane). Suppose we have already evaluated the shading equation for the vertices having resultant intensities I_1, I_2 and I_3, usually on representative wavelengths of red, green and blue light. The color or (R, G, B) values of the inner pixels are determined by linear approximation from the vertex colors. This approximation should be carried out separately for each wavelength.
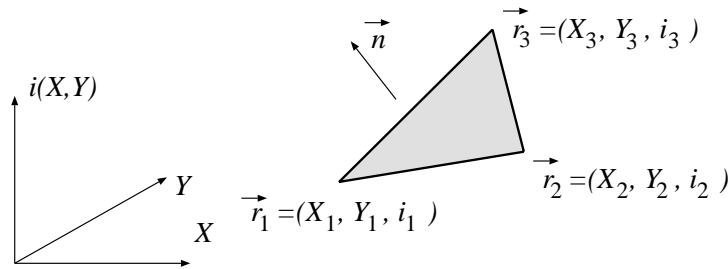
Figure 7.3: Linear interpolation in color space

Let i be the alias of any of I^{red}, I^{green} or I^{blue}. The function $i(X, Y)$ of the pixel coordinates described in figure 7.3 forms a plane through the vertex points $\vec{r}_1 = (X_1, Y_1, i_1)$, $\vec{r}_2 = (X_2, Y_2, i_2)$ and $\vec{r}_3 = (X_3, Y_3, i_3)$ in (X, Y, i) space. For notational convenience, we shall assume that $Y_1 \leq Y_2 \leq Y_3$ and (X_2, Y_2) is on the left side of the $[(X_1, Y_1); (X_3, Y_3)]$ line, looking at the triangle from the camera position. The equation of this plane is:

$$\vec{n} \cdot \vec{r} = \vec{n} \cdot \vec{r}_1 \quad \text{where} \quad \vec{n} = (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1). \tag{7.7}$$

Denoting the constant $\vec{n} \cdot \vec{r}_1$ by C, and expressing the equation in scalar form substituting the coordinates of the normal of the plane, $\vec{n} = (n_X, n_Y, n_i)$, the function $i(X, Y)$ has the following form:

$$i(X, Y) = \frac{C - n_X \cdot X - n_Y \cdot Y}{n_i}. \tag{7.8}$$

The computational requirement of two multiplications, two additions and a division can further be decreased by the incremental concept (recall section 2.3 on hardware realization of graphics algorithms).

Expressing $i(X + 1, Y)$ as a function of $i(X, Y)$ we get:

$$i(X+1, Y) = i(X, Y) + \frac{\partial i(X, Y)}{\partial X} \cdot 1 = i(X, Y) - \frac{n_X}{n_i} = i(X, Y) + \delta i_X. \tag{7.9}$$
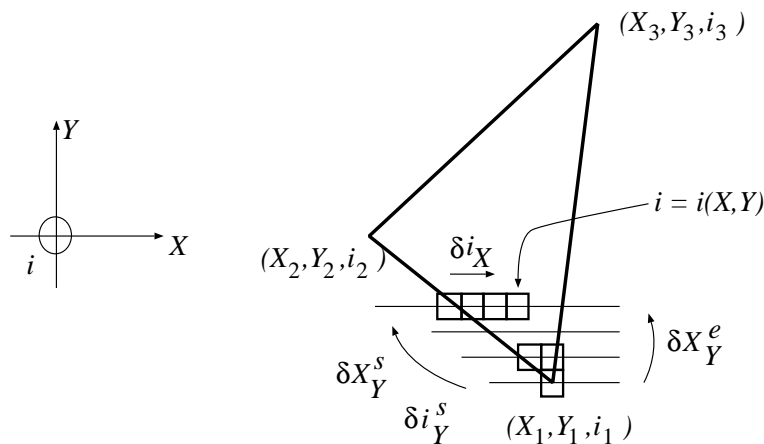
Figure 7.4: Incremental concept in Gouraud shading

Since δi_X does not depend on the actual X, Y coordinates, it has to be evaluated once for the polygon. Inside a scan-line, the calculation of a pixel color requires a single addition for each color coordinate according to equation 7.9. Concerning the X and i coordinates of the boundaries of the scan-lines, the incremental concept can also be applied to express the starting and ending pixels.

Since i and X vary linearly along the edge of the polygon, equations 2.33, 2.34 and 2.35 result in the following simple expressions in the range of $Y_1 \leq Y \leq Y_2$, denoting K_s by X_{start} and K_e by X_{end}, and assuming that the triangle is left oriented as shown in figure 7.4:

$$
\begin{aligned}
X_{\text{start}}(Y+1) &= X_{\text{start}}(Y) + \frac{X_2 - X_1}{Y_2 - Y_1} = X_{\text{start}}(Y) + \delta X_Y^s \\
X_{\text{end}}(Y+1) &= X_{\text{end}}(Y) + \frac{X_3 - X_1}{Y_3 - Y_1} = X_{\text{end}}(Y) + \delta X_Y^e \\
i_{\text{start}}(Y+1) &= i_{\text{start}}(Y) + \frac{i_2 - i_1}{Y_2 - Y_1} = i_{\text{start}}(Y) + \delta i_Y^s \qquad (7.10)
\end{aligned}
$$

The last equation represents in fact three equations, one for each color coordinate, (R, G, B). For the lower part of the triangle in figure 7.4, the incremental algorithm is then:

$X_{\text{start}} = X_1 + 0.5$; $X_{\text{end}} = X_1 + 0.5$;
$R_{\text{start}} = R_1 + 0.5$; $G_{\text{start}} = G_1 + 0.5$; $B_{\text{start}} = B_1 + 0.5$;
for $Y = Y_1$ **to** Y_2 **do**
    $R = R_{\text{start}}$; $G = G_{\text{start}}$; $B = B_{\text{start}}$;
    for $X = \text{Trunc}(X_{\text{start}})$ **to** $\text{Trunc}(X_{\text{end}})$ **do**
        write($X, Y, \text{Trunc}(R), \text{Trunc}(G), \text{Trunc}(B)$);
        $R \mathrel{+}= \delta R_X$; $G \mathrel{+}= \delta G_X$; $B \mathrel{+}= \delta B_X$;
    endfor
    $X_{\text{start}} \mathrel{+}= \delta X_Y^s$; $X_{\text{end}} \mathrel{+}= \delta X_Y^e$;
    $R_{\text{start}} \mathrel{+}= \delta R_Y^s$; $G_{\text{start}} \mathrel{+}= \delta G_Y^{\prime s}$; $B_{\text{start}} \mathrel{+}= \delta B_Y^s$;
endfor

Having represented the numbers in a fixed point format, the derivation of the executing hardware of this algorithm is straightforward by the methods outlined in section 2.3 (on hardware realization of graphics algorithms). Note that this algorithm generates a part of the triangle below Y_2 coordinates. The same method has to be applied again for the upper part.

Recall that the very same approach was applied to calculate the Z coordinate in the z-buffer method. Because of their algorithmic similarity, the same hardware implementation can be used to compute the Z coordinate, and the R, G, B color coordinates.

The possibility of hardware implementation makes Gouraud shading very attractive and popular in advanced graphics workstations, although it has
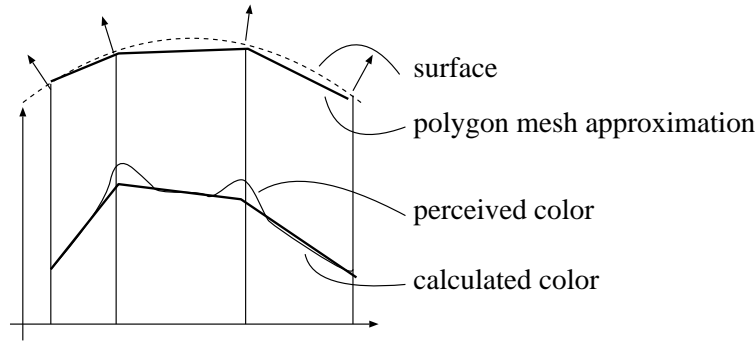
Figure 7.5: Mach banding

several severe drawbacks. It does not allow shadows, texture and bump mapping in its original form, and introduces an annoying artifact called **Mach banding** (figure 7.5). Due to linear approximation in color space, the color is a continuous, but not differentiable function. The human eye, however, is sensitive to the drastic changes of the derivative of the color, overemphasizing the edges of the polygon mesh, where the derivative is not continuous.

7.6   Phong shading

In Phong shading only the surface normal is approximated from the real surface normals in the vertices of the approximating polygon; the shading equation is evaluated for each pixel. The interpolating function of the normal vectors is linear:

$$\begin{aligned}
n_X &= a_X \cdot X + b_X \cdot Y + c_X, \\
n_Y &= a_Y \cdot X + b_Y \cdot Y + c_Y, \\
n_Z &= a_Z \cdot X + b_Z \cdot Y + c_Z.
\end{aligned} \tag{7.11}$$

Constants $a_X, \ldots, c_Z$ can be determined by similar considerations as in Gouraud shading from the normal vectors at the vertices of the polygon (triangle). Although the incremental concept could be used again to reduce the number of multiplications in this equation, it is not always worth doing, since the shading equation requires many expensive computational steps

which mean that this computation is negligible in terms of the total time required.

Having generated the approximation of the normal to a surface visible in a given pixel, the complete rendering equation is applied:

$$I^{\text{out}} = I_e + k_a \cdot I_a + \sum_l^{n_l} r_l \cdot I_l \cdot k_d \cdot \max\{(\vec{N} \cdot \vec{L}), 0\}+$$

$$\sum_l^{n_l} r_l \cdot I_l \cdot k_s \cdot \max\{[2(\vec{N} \cdot \vec{H})^2 - 1]^n, 0\} \qquad (7.12)$$

Recall that dot products, such as $\vec{N} \cdot \vec{L}$, must be evaluated for vectors in the world coordinate system, since the viewing transformation may alter the angle between vectors. For directional lightsources this poses no problem, but for positional and flood types the point corresponding to the pixel in the world coordinate system must be derived for each pixel. To avoid screen and world coordinate system mappings on the pixel level, the corresponding (x, y, z) world coordinates of the pixels inside the polygon are determined by a parallel and independent linear interpolation in world space. Note that this is not accurate for perspective transformation, since the homogeneous division of perspective transformation destroys equal spacing, but this error is usually not noticeable on the images.

Assuming only ambient and directional lightsources to be present, the incremental algorithm for half of a triangle is:

$$X_{\text{start}} = X_1 + 0.5;\ X_{\text{end}} = X_1 + 0.5;$$
$$\vec{N}_{\text{start}} = \vec{N}_1;$$
for $Y = Y_1$ **to** Y_2 **do**
$\quad \vec{N} = \vec{N}_{\text{start}};$
$\quad$ **for** $X = \text{Trunc}(X_{\text{start}})$ **to** $\text{Trunc}(X_{\text{end}})$ **do**
$\quad\quad (R, G, B) = \text{ShadingModel}(\ \vec{N}\ );$
$\quad\quad$ **write**$(\ X, Y, \text{Trunc}(R), \text{Trunc}(G), \text{Trunc}(B)\ );$
$\quad\quad \vec{N}\ += \delta\vec{N}_X;$
$\quad$ **endfor**
$\quad X_{\text{start}}\ += \delta X_Y^s;\ X_{\text{end}}\ += \delta X_Y^e;$
$\quad \vec{N}_{\text{start}}\ += \delta\vec{N}_Y^s;$
endfor

The rendering equation used for Phong shading is not appropriate for
incremental evaluation in its original form. For directional and ambient
lightsources, however, it can be approximated by a two-dimensional Taylor
series, as proposed by Bishop [BW86], which in turn can be calculated incre-
mentally with five additions and a non-linear function evaluation typically
implemented by a pre-computed table in the computer memory.

The coefficients of the shading equation, k_a, k_d, k_s and n can also be
a function of the point on the surface, allowing **textured surfaces** to be
rendered by Phong shading. In addition it is possible for the approximated
surface normal to be perturbed by a normal vector variation function caus-
ing the effect of bump mapping (see chapter 12).

Chapter 8

z-BUFFER, GOURAUD-SHADING WORKSTATIONS

As different shading methods and visibility calculations have diversified the image generation, many different alternatives have come into existence for their implementation. This chapter will focus on a very popular solution using the z-buffer technique for hidden surface removal, and Gouraud shading for color computation.

The main requirements of an advanced workstation of this category are:

- The workstation has to generate both 2D and 3D graphics at the speed required for interactive manipulation and real-time animation.

- At least wire-frame, hidden-line and solid — Gouraud and constant shaded — display of 3D objects broken down into polygon lists must be supported. Some technique has to be applied to ease interpretation of wire frame images.

- Both parallel and perspective projections are to be supported.

- Methods reducing the artifacts of sampling and quantization are needed.

- The required resolution is over 1000×1000 pixels, the frame buffer must have at least 12, but preferably 24 bits/pixel to allow for true

color mode and double buffering for animation. The z-buffer must have at least 16 bits/pixel.

8.1   Survey of wire frame image generation

The dataflow model of the wire frame image generation in a system applying z-buffer and Gouraud shading is described in figure 8.1. The **decomposition** reads the internal model and converts it to a wire-frame representation providing a list of edges defined by the two endpoints in the local modeling coordinate system for each object. The points are transformed first by the **modeling transformation** $\mathbf{T_M}$ to generate the points in the common world coordinate system. The modeling transformation is set before processing each object. From the world coordinate system the points are transformed again to the screen coordinate system for parallel projection and to the 4D homogeneous coordinate system for perspective projection by a **viewing transformation** $\mathbf{T_V}$. Since the matrix multiplications needed by the modeling and viewing transformations can be concatenated, the transformation from the local modeling coordinates to the screen or to the 4D homogeneous coordinate system can be realized by a single matrix multiplication by a composite transformation matrix $\mathbf{T_C = T_M \cdot T_V}$.

For parallel projection, the complete clipping is to be done in the screen coordinate system by, for example, the 3D version of the Cohen–Sutherland clipping algorithm. For perspective projection, however, at least the **depth clipping** phase must be carried out before the homogeneous division, that is in the 4D homogeneous coordinate system, then the real 3D coordinates have to be generated by the homogeneous division, and **clipping against the side faces** should be accomplished if this was not done in the 4D homogeneous space.

The structure of the screen coordinate system is independent of the type of projection, the X, Y coordinates of a point refer to the projected coordinates in pixel space, and Z is a monotonously increasing function of the distance from the camera. Thus the **projection** is trivial, only the X, Y coordinates have to be extracted.

The next phase of the image generation is **scan conversion**, meaning the selection of those pixels which approximate the given line segment and also the color calculation of those pixels. Since pixels correspond to the integer

internal model

model decomposition

Line segments $(x_L, y_L, z_L, 1)_{1,2}$ in local coordinates

$$\mathbf{T_C} = \left\{ \begin{array}{l} \cdot \mathbf{T_M} \\[2ex] \cdot \mathbf{T_{VIEW}} = \mathbf{T_{uvw}^{-1}} \cdot \mathbf{T_{eye}} \cdot \mathbf{T_{shear}} \cdot \mathbf{T_{norm}} \end{array} \right.$$

Line segment: $(X_h, Y_h, Z_h, h)_{1,2}$ in 4D homogenous system

depth clipping

homogenous division
$X = X_h/h; \ Y = Y_h/h; \ Z = Z_h/h$

side face clipping

scan conversion / depth cueing

Pixel series: $(X, Y, i)|_{1,2,\ldots n}$

pixel operations

frame buffer write

pixel data in frame buffer
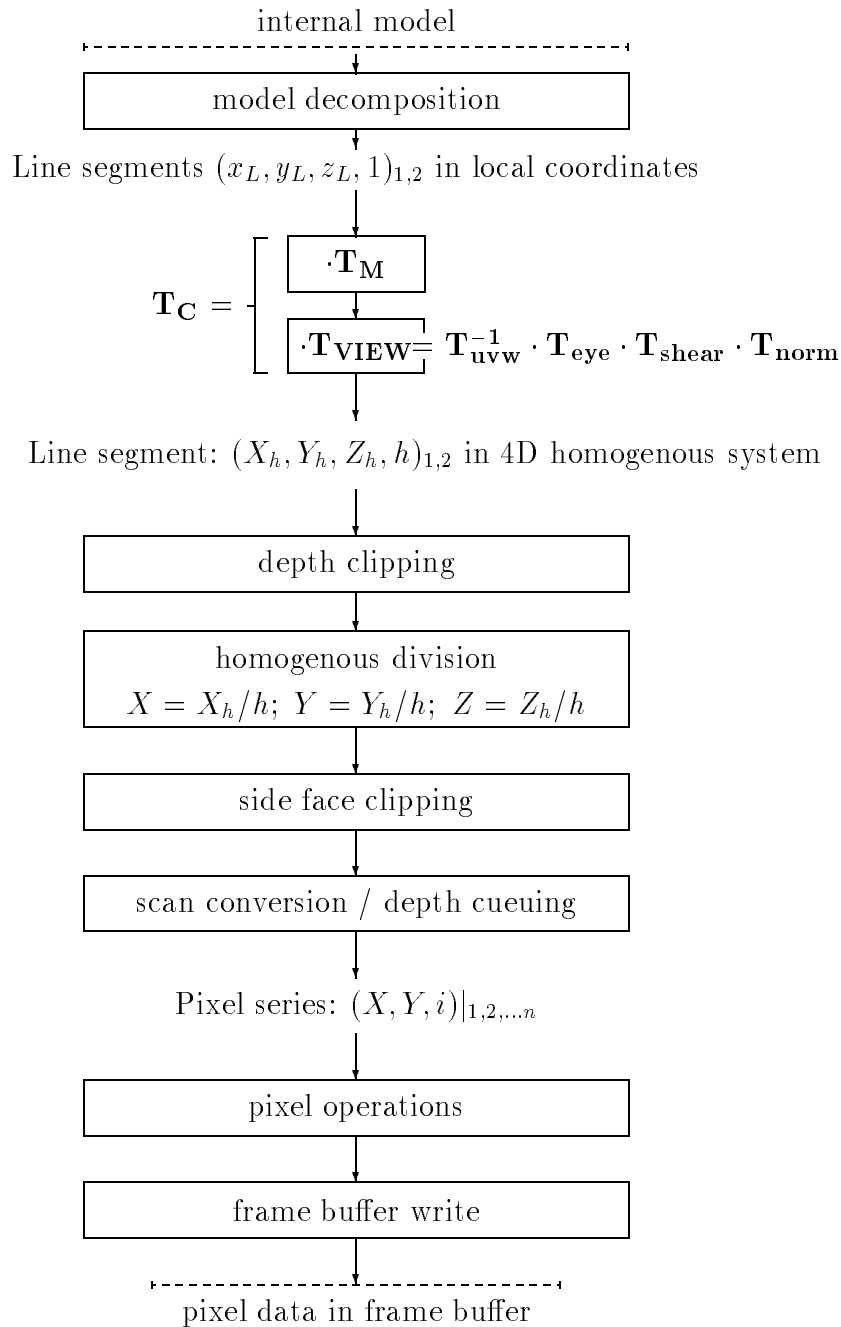
Figure 8.1: *Data flow of wire frame image synthesis (perspective projection)*

grid of pixel space, and scan conversion algorithms usually rely on the integer representation of endpoint coordinates, the coordinates are truncated or rounded to integers.

Concerning color calculation, or **shading**, it is not worth working with sophisticated shading and illumination models when the final image is wireframe. The simple assumption that all pixels of the vectors have the same color, however, is often not satisfactory, because many lines crossing each other may confuse the observer, inhibiting reconstruction of the 3D shape in his mind. The understandability of wire-frame images, however, can be improved by a useful trick, called **depth cueing**, which uses more intense colors for points closer to the camera, while the color decays into the background as the distance of the line segments increases, corresponding to a simplified shading model defining a single lightsource in the camera position.

The outcome of scan-conversion is a series of pixels defined by the integer coordinates X_p, Y_p and the pixel color i. Before writing the color information of the addressed pixel into the raster memory various operations can be applied to the individual pixels. These **pixel level operations** may include the reduction of the quantization effects by the means of **dithering**, or arithmetic and logic operations with the pixel data already stored at the X_p, Y_p location. This latter procedure is called the **raster operation**.

Anti-aliasing techniques, for example, require the weighted addition of the new and the already stored colors. A simple exclusive OR (XOR) operation, on the other hand, allows the later erasure of a part of the wire-frame image without affecting the other part, based on the identity $(A \oplus B) \oplus B = A$. Raster operations need not only the generated color information, but also the color stored in the frame buffer at the given pixel location, thus an extra frame buffer read cycle is required by them.

The result of pixel level operations is finally written into the **frame buffer memory** which is periodically scanned by the video display circuits which generate the color distribution of the display according to the stored frame buffer data.

8.2   Survey of shaded image generation

The dataflow model of the shaded image generation in a **z-buffer**, **Gouraud shading** system is described in figure 8.2.
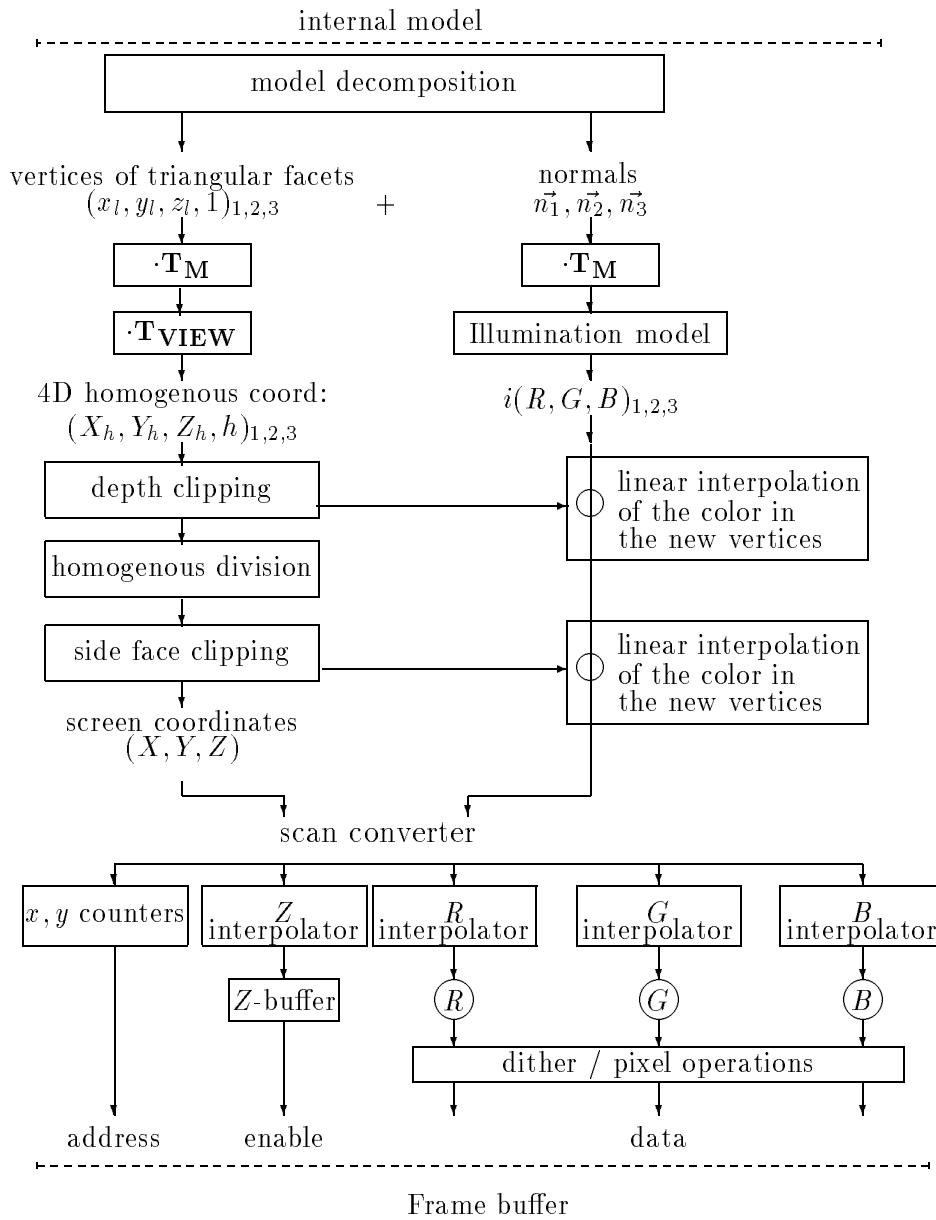
Figure 8.2: Data flow of shaded image synthesis

Now the **decomposition** reads the internal model and converts it to a polygon list representation defining each polygon by its vertices in the local modeling coordinate system for each object. To provide the necessary information for shading, the real normals of the surfaces approximated by polygon meshes are also computed at polygon vertices. The vertices are transformed first by the modeling transformation then by the viewing transformation by a single matrix multiplication with the composite transformation matrix. Normal vectors, however, are transformed to the world coordinate system, because that is a proper place for illumination calculation. Coordinate systems after shearing and perspective transformation are not suitable, since they do not preserve angles, causing incorrect calculation of dot products. According to the concept of Gouraud shading, the **illumination equation** is evaluated for each vertex of the polygon mesh approximating the real surfaces, using the real surface normals at these points. **Depth cueing** can also be applied to shaded image generation if the illumination equation is modified to attenuate the intensity proportionally to the distance from the camera. The linear decay of the color at the internal pixels will be guaranteed by linear interpolation of the Gouraud shading.

Similarly to wire frame image generation, the complete clipping is to be done in the screen coordinate system for parallel projection. An applicable clipping algorithm is the 3D version of the Sutherland-Hodgman polygon clipping algorithm. For perspective projection, however, at least the **depth clipping** phase must be done before homogeneous division, that is in the 4D homogeneous coordinate system, then the real 3D coordinates have to be generated by homogeneous division, and **clipping against the side faces** should be accomplished if this was not done in 4D homogeneous space.

After the trivial **projection** in the screen coordinate system, the next phase of image generation is **scan conversion** meaning the selection of those pixels which approximate the given polygon and also the interpolation of pixel colors from the vertex colors coming from the illumination formulae evaluated in the world coordinate system. Since pixels correspond to the integer grid of the pixel space, and scan conversion algorithms usually rely on the integer representation of endpoint coordinates, the coordinates are truncated or rounded to integers. The z-buffer visibility calculation method resolves the hidden surface problem during the scan conversion comparing the Z-coordinate of each pixel and the value already stored in the z-buffer

memory. Since the transformation to the screen coordinate system has been carefully selected to preserve planes, the Z-coordinate of an inner point can be determined by linear interpolation of the Z-coordinates of the vertices. This Z-interpolation and the color interpolation for the R, G and B components are usually executed by a digital network. Since in hardware implementations the number of variables is not flexible, polygons must be decomposed into triangles defined by three vertices before the interpolation.

The pixel series resulting from the polygon or **facet** scan conversion can also go through pixel level operations before being written into the frame buffer. In addition to dithering and arithmetic and logic raster operations, the illusion of transparency can also be generated by an appropriate pixel level method which is regarded as the application of **translucency patterns**. The final colors are eventually written into the frame buffer memory.

8.3 General system architecture

Examining the tasks to be executed during image generation from the point of view of data types, operations, speed requirements and the allocated hardware resources, the complete pipeline can be broken down into the following main stages:

1. *Internal model access and primitive decomposition.* This stage should be as flexible as possible to incorporate a wide range of models. The algorithms are also general, thus some general purpose processor must be used to run the executing programs. This processor will be called the **model access processor** which is a sort of interface between the graphics subsystem and the rest of the system. The model access and primitive decomposition step needs to be executed once for an interactive manipulation sequence and for animation which are the most time critical applications. Thus, if there is a temporary memory to store the primitives generated from the internal model, then the speed requirement of this stage is relatively modest. This buffer memory storing graphics primitives is usually called the **display list memory**. The display list is the low level representation of the model to be rendered on the computer screen in conjunction with the camera and display parameters. Display lists are interpreted and processed by a so-called **display list processor** which controls the functional

elements taking part in the image synthesis. Thus, the records of display lists can often be regarded as operation codes or instructions to a special purpose processor, and the content of the display list memory as an executable program which generates the desired image.

2. *Geometric manipulations* including transformation, clipping, projection and illumination calculation. This stage deals with geometric primitives defined by points represented by coordinate triples. The coordinates are usually floating point numbers to allow flexibility and to avoid rounding errors. At this stage fast, but simple floating point arithmetic is needed, including addition, multiplication, division and also square roots for shading calculations, but the control flow is very simple and there is no need for accessing large data structures. A cost effective realization of this stage may contain floating point signal processors, bit-slice ALUs or floating point co-processors. The hardware unit responsible for these tasks is usually called the **geometry engine**, although one of its tasks, the illumination calculation, is not a geometric problem. The geometry engines of advanced workstations can process about 1 million points per second.

3. *Scan-conversion, z-buffering and pixel level operations.* These tasks process individual pixels whose number can exceed 1 million for a single image. This means that the time available for a single pixel is very small, usually several tens of nanoseconds. Up to now commercial programmable devices have not been capable of coping with such a speed, thus the only alternatives were special purpose digital networks, or high degree parallelization. However, recently very fast RISC processors optimized for graphics have appeared, implementing internal parallelization and using large cache memories to decrease significantly the number of memory cycles to fetch instructions. A successful representative of this class of processors is the *intel 860* microprocessor [Int89] [DRSK92] which can be used not only for scan conversion, but also as a geometry engine because of its appealing floating point performance. At the level of scan-conversion, z-buffering and pixel operations, four sub-stages can be identified. **Scan conversion** is responsible for the change of the representation from geometric to pixel. The hardware unit executing this task is called the **scan converter**.

The **z-buffering hardware** includes both the comparator logic and the **z-buffer** memory, and generates an enabling signal to overwrite the color stored in the frame buffer while it is updating the z-value for the actual pixel. Thus, to process a single pixel, the z-buffer memory needs to be accessed for a read and an optional write cycle. Comparing the speed requirements — several tens of nanosecond for a single pixel —, and the cycle time of the memories which are suitable to realize several megabytes of storage — about a hundred nanoseconds —, it becomes obvious that some special architecture is needed to allow the read and write cycles to be accomplished in time. The solutions applicable are similar to those used for frame buffer memory design. Pixel level operations can be classified according to their need of color information already stored in the frame buffer. Units carrying out **dithering** and generating translucency patterns do not use the colors already stored at all. **Raster operations**, on the other hand, produce a new color value as a result of an operation on the calculated and the already stored colors, thus they need to access the frame buffer.

4. *Frame buffer* storage. Writing the generated pixels into the **frame buffer memory** also poses difficult problems, since the cycle time of commercial memories are several times greater than the expected few tens of nanoseconds, but the size of the frame buffer — several megabytes — does not allow for the usage of very high speed memories. Fortunately, we can take advantage of the fact that pixels are generated in a coherent way by image synthesis algorithms; that is if a pixel is written into the memory the next one will probably be that one which is adjacent to it. The frame buffer memory must be separated into **channels**, allocating a separate bus for each of them in such a way that on a scan line adjacent pixels correspond to different channels. Since this organization allows for the parallel access of those pixels that correspond to different channels, this architecture approximately decreases the access time by a factor of the number of channels for coherent accesses.

5. *The display of the content of the frame buffer* needs **video display hardware** which scans the frame buffer 50, 60 or 70 times each second

and produces the analog R, G and B signals for the color monitor. Since the frame buffer contains about 10^6 number of pixels, the time available for a single pixel is about 10 nanoseconds. This speed requirement can only be met by special hardware solutions. A further problem arises from the fact that the frame buffer is a double access memory, since the image synthesis is continuously writing new values into it while the video hardware is reading it to send its content to the color monitor. Both directions have critical timing requirements — ten nanoseconds and several tens of nanoseconds — higher than would be provided by a conventional memory architecture. Fortunately, the display hardware needs the pixel data very coherently, that is, pixels are accessed one after the other from left to right, and from top to bottom. Using this property, the frame buffer row being displayed can be loaded into a shift register which in turn rolls out the pixels one-by-one at the required speed and without accessing the frame buffer until the end of the current row. The series of consecutive pixels may be regarded as addresses of a **color lookup table** to allow a last transformation before **digital-analog conversion**. For indexed color mode, this lookup table converts the color indices (also called **pseudo-colors**) into R, G, B values. For **true color mode**, on the other hand, the R, G, B values stored in the frame buffer are used as three separate addresses in three **lookup tables** which are responsible for γ-**correction**. The size of these lookup tables is usually modest — typically $3 \times 256 \times 8$ bits — thus very high speed memories having access times less than 10 nanoseconds can be used. The outputs of the lookup tables are converted to analog signals by three **digital-to-analog converters**.

Summarizing, the following hardware units can be identified in the graphics subsystem of an advanced workstation of the discussed category: model access processor, display list memory, display list processor, geometry engine, scan converter, z-buffer comparator and controller, z-buffer memory, dithering and translucency unit, **raster operation ALUs**, frame buffer memory, video display hardware, lookup tables, D/A converters. Since each of these units is responsible for a specific stage of the process of the image generation, they should form a pipe-line structure. Graphics subsystems generating the images are thus called as the **output or image generation**
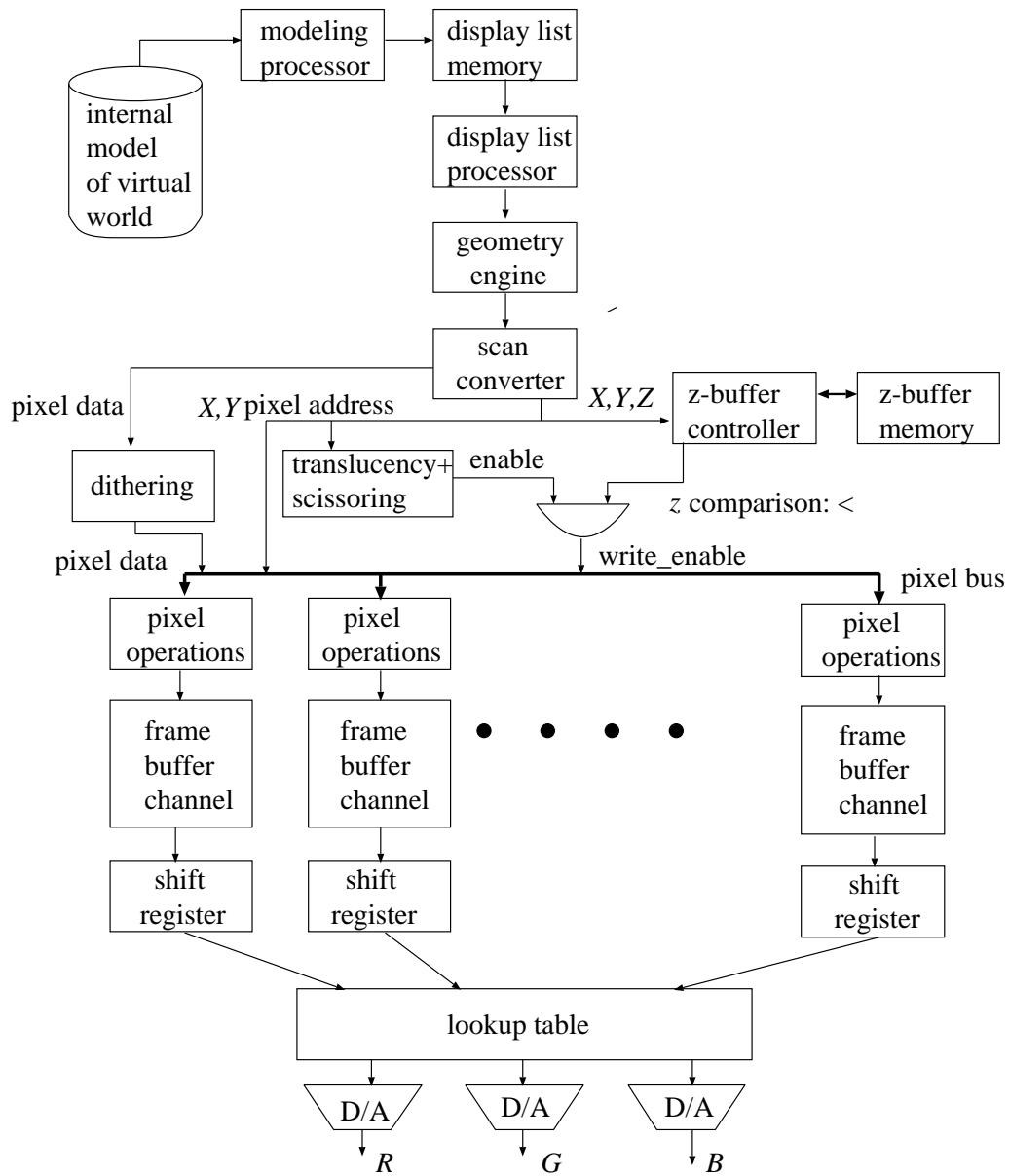
Figure 8.3: Architecture of z-buffer, Gouraud-shading graphics systems

pipelines. Interaction devices usually form a similar structure, which is called the **input pipeline**.

In the output pipeline the units can be grouped into two main subsystems: a high-level subsystem which works with geometric information and a low-level subsystem which deals with pixel information.

8.4   High-level subsystem

The high-level subsystem consists of the model access and display list processors, the display list memory and the geometry engine.

The model access processor is always, the display processor is often, a general purpose processor. The display list processor which is responsible for controlling the rest of the display pipeline can also be implemented as a special purpose processor executing the program of the display list. The display list memory is the interface between the model access processor and the display list processor, and thus it must have double access organization. The advantages of display list memories can be understood if the case of an animation sequence is considered. The geometric models of the objects need to be converted to display list records or instructions only once before the first image. The same data represented in an optimal way can be used again for each frame of the whole sequence, the model access processor just modifies the transformation matrices and viewing parameters before triggering the display list processor. Thus, both the computational burden of the model access processor and the communication between the model access and display list processors are modest, allowing the special purpose elements to utilize their maximum performance.

The display list processor interprets and executes the display lists by either realizing the necessary operations or by providing control to the other hardware units. A lookup table set instruction, for example, is executed by the display list processor. Encountering a *DRAWLINE* instruction, on the other hand, it gets the geometry engine to carry out the necessary transformation and clipping steps, and forces the scan converter to draw the screen space line at the points received from the geometry engine. Thus, the geometry engine can be regarded as the floating-point and special instruction set co-processor of the display list processor.

8.5   Low-level subsystem

8.5.1   Scan conversion hardware

Scan conversion of lines

The most often used line generators are the implementations of Bresenham's incremental algorithm that uses simple operations that can be directly implemented by combinational elements and does not need division and other complicated operations during initialization. The basic algorithm can generate the pixel addresses of a 2D digital line, therefore it must be extended to produce the Z coordinates of the internal pixels and also their color intensities if depth cueing is required. The Z coordinates and the pixel colors ought to be generated by an incremental algorithm to allow for easy hardware implementation. In order to derive such an incremental formula, the increment of the Z coordinate and the color is determined. Let the 3D screen space coordinates of the two end points of the line be $[X_1, Y_1, Z_1]$ and $[X_2, Y_2, Z_2]$, respectively and suppose that the z-buffer can hold values in the range $[0 \ldots Z_{\max}]$. Depth cueing requires the attenuation of the colors by a factor proportional to the distance from the camera, which is represented by the Z coordinate of the point. Assume that the intensity factor of depth cueing is $C_{\max}$ for $Z = 0$ and $C_{\min}$ for $Z_{\max}$. The number of pixels composing this digital line is:

$$L = \max\{|X_2 - X_1|, |Y_2 - Y_1|\}. \tag{8.1}$$

Since Z varies linearly along the line, the difference of the Z coordinates of two consecutive pixel centers is:

$$\Delta Z = \frac{Z_2 - Z_1}{L}. \tag{8.2}$$

Let I stand for any of the line's three color coordinates R, G, B. The perceived color, taking into account the effect of depth cueing, is:

$$I^*(Z) = I \cdot C(Z) = I \cdot (C_{\max} - \frac{C_{\max} - C_{\min}}{Z_{\max}} \cdot Z). \tag{8.3}$$

The difference in color of the two pixel centers is:

$$\Delta I = \frac{I^*(Z_2) - I^*(Z_1)}{L}. \tag{8.4}$$
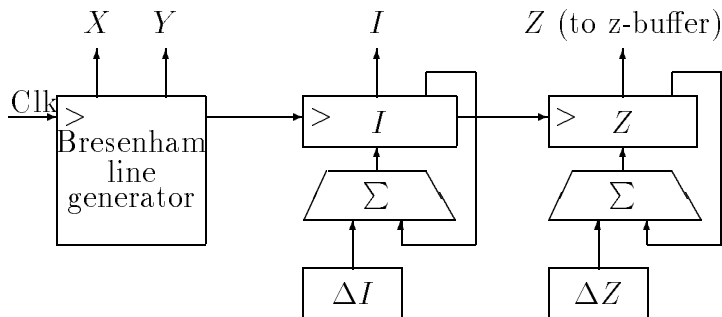
Figure 8.4: Hardware to draw depth cueing lines

For easy hardware realization, Z and I^* should be computed by integer additions. Examining the formulae for ΔZ and ΔI^*, we will see that they are non-integers and not necessarily positive. Thus, some signed fixed point representation must be selected for storing Z and I^*. The calculation of the Z coordinate and color I^* can thus be integrated into the internal loop of the Bresenham's algorithm:

3D_BresenhamLine $(X_1, Y_1, Z_1, X_2, Y_2, Z_2, I)$
  Initialize a 2D Bresenham's line generator(X_1, Y_1, X_2, Y_2);
  $L = \max\{|X_2 - X_1|, |Y_2 - Y_1|\}$;
  $\Delta Z = (Z_2 - Z_1)/L$;
  $\Delta I = I \cdot ((C_{\min} - C_{\max}) \cdot (Z_2 - Z_1))/(Z_{\max} \cdot L)$;
  $Z = Z_1 + 0.5$;
  $I^* = I \cdot (C_{\max} - (Z_1 \cdot (C_{\max} - C_{\min}))/Z_{\max}) + 0.5$;
  for $X = X_1$ **to** X_2 **do**
   Iterate Bresenham's algorithm$(\ X, Y\ )$;
   $I^* \mathrel{+}= \Delta I$; $Z \mathrel{+}= \Delta Z$; $z = \text{Trunc}(Z)$;
   if Zbuffer$[X, Y] > z$ **then**
    Write Zbuffer(X, Y, z);
    Write frame buffer$(X, Y, \text{Trunc}(I^*))$;
   endif
  endfor

The z-buffer check is only necessary if the line drawing is mixed with shaded image generation, and it can be neglected when the complete image is wire frame.

Scan-conversion of triangles

For hidden surface elimination the z-buffer method can be used together with **Gouraud shading** if a shaded image is needed or with constant shading if a **hidden-line** picture is generated. The latter is based on the recognition that hidden lines can be eliminated by a special version of the z-buffer hidden surface algorithm which draws polygons generating their edges with the line color and filling their interior with the color of the background. In the final result the edges of the visible polygons will be seen, which are, in fact, the visible edges of the object. **Constant shading**, on the other hand, is a special version of the linear interpolation used in Gouraud shading with zero color increments. Thus the linear color interpolator can also be used for the generation of constant shaded and hidden-line images. The linear interpolation over a triangle is a two-dimensional interpolation over the pixel coordinates X and Y, which can be realized by a digital network as discussed in subsection 2.3.2 on hardware realization of multi-variate functions. Since a color value consists of three scalar components — the R, G and B coordinates — and the internal pixels' Z coordinates used for z-buffer checks are also produced by a linear interpolation, the interpolator must generate four two-variate functions. The applicable incremental algorithms have been discussed in section 6.3 (z-buffer method) and in section 7.5 (Gouraud shading). The complete hardware system is shown in figure 8.5.

8.5.2 z-buffer

The z-buffer consists of a Z-comparator logic and the memory subsystem. As has been mentioned, the memory must have a special organization to allow higher access speed than provided by individual memory chips when they are accessed coherently; that is in the order of subsequent pixels in a single pixel row. The same memory design problem arises in the context of the frame buffer, thus its solution will be discussed in the section of the frame buffer.
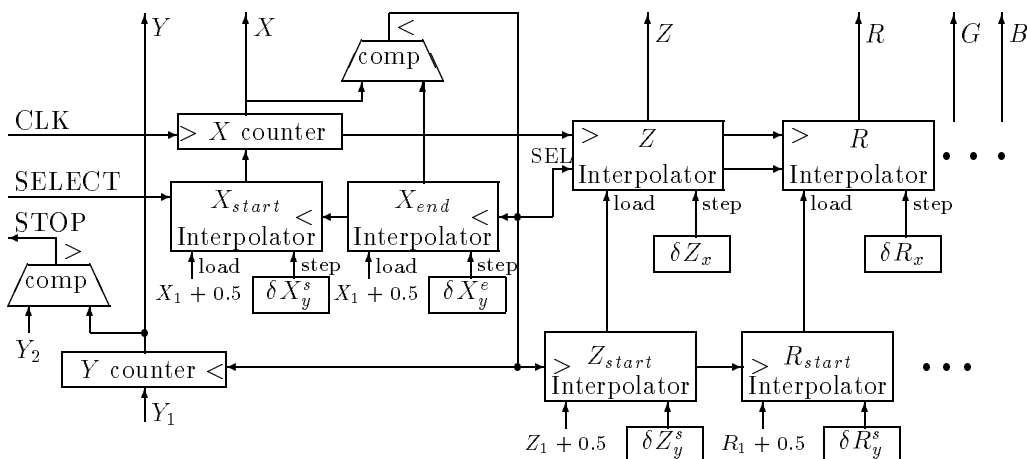
Figure 8.5: Scan converter for rendering triangles

The Z-comparator consists of a comparator element and a temporary register to hold the Z value already stored in the z-buffer. A comparison starts by loading the Z value stored at the X, Y location of the z-buffer into the temporary register. This is compared with the new Z value, resulting in an enabling signal that is true (enabled) if the new Z value is smaller than the one already stored. The Z-comparator then tries to write the new value into the z-buffer controlled by the enabling signal. If the enabling signal is true, then the write operation will succeed, otherwise the write operation will not alter the content of the z-buffer. The same enabling signal is used to enable or disable rewriting the content of the frame buffer to make the z-buffer algorithm complete.

8.5.3 Pixel-level operation

There are two categories of pixel-level operations: those which belong to the first category are based on only the new color values, and those which generate the final color from the color coming from the scan converter and the color stored in the frame buffer fall into the second category. The first category is a post-processing step of the scan conversion, while the second

is a part of the frame buffer operation. Important examples of the post-processing class are the transparency support, called the **translucency generator**, the dithering hardware and the **overlay management**.

Support of translucency and dithering

As has been stated, transparency can be simulated if the surfaces are written into the frame buffer in order of decreasing distance from the camera and when a new pixel color is calculated, a weighted sum is computed from the new color and the color already stored in the frame buffer. The weight is defined by the transparency coefficient of the object. This is obviously a pixel operation. The dependence on the already stored color value, however, can be eliminated if the weighting summation is not restricted to a single pixel, and the low-pass filtering property of the human eye is also taken into consideration.

Suppose that when a new surface is rendered some of its spatially uniformly selected pixels are not written into the frame buffer memory. The image will contain pixel colors from the new surface and from the previously rendered surface — which is behind the last surface — that are mixed together. The human eye will filter this image and will produce the perception of some mixed color from the high frequency variations due to alternating the colors of several surfaces.

This is similar to looking through a fine net. Since in the holes of the net the world behind the net is visible, if the net is fine enough, the observer will have the feeling that he perceives the world through a transparent object whose color is determined by the color of the net, and whose transparency is given by the relative size of the holes in the net.

The implementation of this idea is straightforward. Masks, called **translucency patterns**, are defined to control the effective degree of transparency (the density of the net), and when a surface is written into the frame buffer, the X, Y coordinates of the actual pixel are checked whether or not they select a 0 (a hole) in the mask (net), and the frame buffer write operation is enabled or disabled according to the mask value.

This check is especially easy if the mask is defined as a 4×4 periodic pattern. Let us denote the low 2 bits of X and Y coordinates by $X|_2$ and $Y|_2$ respectively. If the 4×4 translucency pattern is $T[x, y]$, then the bit enabling the frame buffer write is $T[X|_2, Y|_2]$.

The hardware generating this can readily be combined with the dithering hardware discussed in subsection 11.5.2 (ordered dithers), as described in figure 8.6.
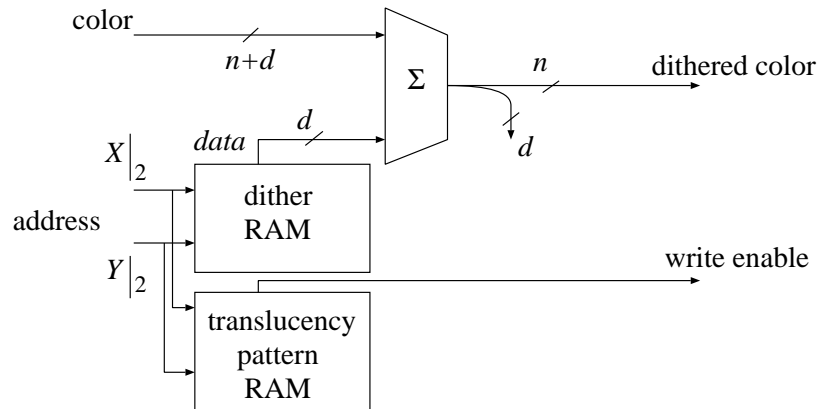
Figure 8.6: Dither and translucency pattern generator

8.5.4   Frame buffer

The frame buffer memory is responsible for storing the generated image in digital form and for allowing the video display hardware to scan it at the speed required for flicker-free display. As stated, the frame buffer is a double access memory, since it must be modified by the drawing engine on the one hand, while it is being scanned by the video display hardware on the other hand. Both access types have very critical speed requirements which exceed the speed of commercial memory chips, necessitating special architectural solutions. These solutions increase the effective access speed for "coherent" accesses, that is for those consecutive accesses which need data from different parts of the memory. The problem of the video refresh access is solved by the application of temporary shift registers which are loaded parallelly, and are usually capable of storing a single row of the image. These shift registers can then be used to produce the pixels at the speed of the display scan (approx. 10 nsec/pixel) without blocking memory access from the drawing engine.
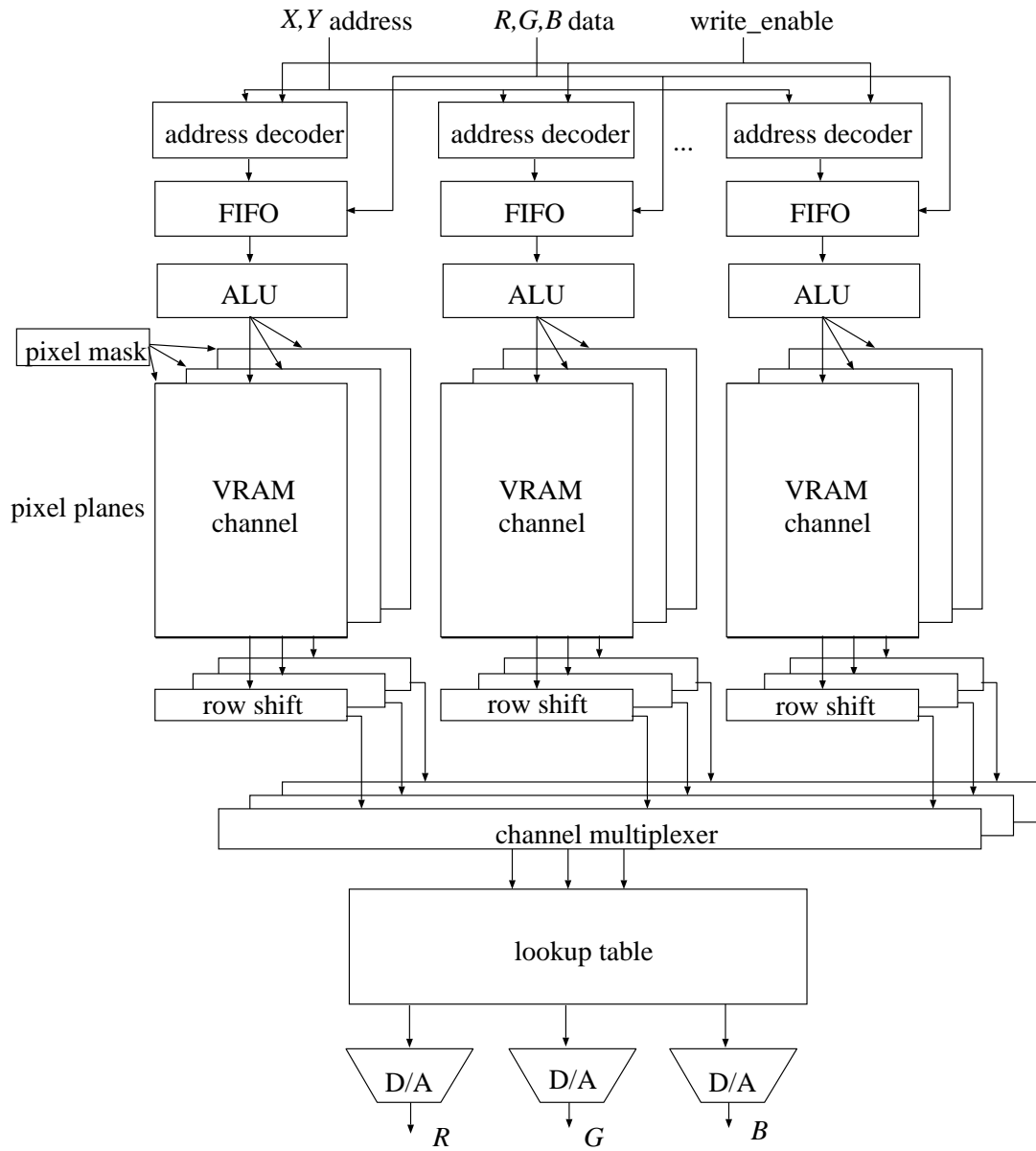
Figure 8.7: Frame buffer architecture

The problem of high speed drawing accesses can be addressed by partitioning the memory into independent channels and adding high-speed temporary registers or FIFOs to these channels. The write operation of these FIFOs needs very little time, and having written the new data into it, a separate control logic loads the data into the frame buffer memory at the speed allowed by the memory chips. If a channel is not accessed very often, then the effective access speed will be the speed of accessing the temporary register of FIFO, but if the pixels of a single channel are accessed repeatedly, then the access time will degrade to that of the memory chips. That is why adjacent pixels are assigned to different channels, because this decreases the probability of repeated accesses for normal drawing algorithms. FIFOs can compensate for the uneven load of different channels up to their capacity.

In addition to these, the frame buffer is also expected to execute arithmetic and logic operations on the new and the stored data before modifying its content. This can be done without significant performance sacrifice if the different channels are given independent ALUs, usually integrated with the FIFOs.

The resulting frame buffer architecture is shown in figure 8.7.

Chapter 9

RECURSIVE RAY TRACING

9.1 Simplification of the illumination model

The light that reaches the eye through a given pixel comes from the surface of an object. The smaller the pixel is, the higher the probability that only one object affects its color, and the smaller the surface element that contributes to the light ray. The energy of this ray consists of three main components. The first component comes from the own emission of the surface. The second component is the energy that the surface reflects into the solid angle corresponding the the pixel, while the third is the light energy propagated by refraction. The origin of the reflective component is either a lightsource (primary reflection) or the surface of another object (secondary, ternary, etc. reflections). The origin of the refracted component is always on the surface of the same object, because this component is going through its interior. We have seen in chapter 3 that the intensity of the reflected light can be approximated by accumulating the following components:

- an ambient intensity I_0, which is the product of the ambient reflection coefficient k_a of the surface and a global ambient intensity I_a assumed to be the same at each spatial point

- a diffuse intensity I_d, which depends on the diffuse reflection coefficient k_d of the surface and the intensity and incidence angle of the light reaching the surface element from any direction

- a specular intensity I_s, which depends on the specular reflection coefficient k_s of the surface and the intensity of the light. In addition, the value is multiplied by a term depending on the angle between the theoretical direction of reflection and the direction of interest and a further parameter n called the specular exponent

- a reflection intensity I_r, which is the product of the (coherent) reflective coefficient k_r of the surface and the intensity of the light coming from the inverse direction of reflection.

Refracted light can be handled similarly.

The following simplifications will be made in the calculations:

- Light rays are assumed to have zero width. This means that they can be treated as lines, and are governed by the laws of geometric optics. The ray corresponding to a pixel of the image can be a line going through any of its points, in practice the ray is taken through its center. A consequence of this simplification is that the intersection of a ray and the surface of an object becomes a single point instead of a finite surface element.

- Diffuse and specular components in the reflected light are considered only for primary reflections; that is, secondary, ternary, etc. incoherent reflections are ignored (these can be handled by the radiosity method). This means that if the diffuse and specular components are to be calculated for a ray leaving a given surface point, then the possible origins are not searched for on the surfaces of other objects, but only the lightsources will be considered.

- When calculating the coherent reflective and refractive components for a ray leaving a given surface point, its origin is searched for on the surface of the objects. Two rays are shot towards the inverse direction of reflection and refraction, respectively, and the first surface points that they intersect are calculated. These rays are called the children of our original ray. Due to multiple reflections and refractions, child

rays can have their own children, and the family of rays corresponding to a pixel forms a binary tree. In order to avoid infinite recurrence, the depth of the tree is limited.

- Incoherent refraction is completely ignored. Implying this would cause no extra difficulties — we could use a very similar model to that for incoherent reflection — but usually there is no practical need for it.
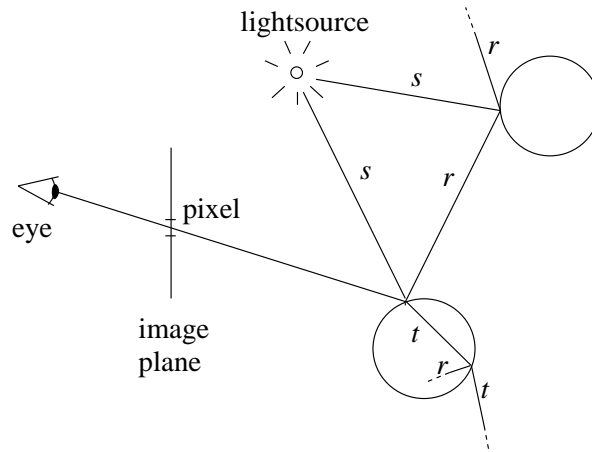
Figure 9.1: Recursive ray tracing

These concepts lead us to recursive ray tracing. Light rays will be *traced backwards* (contrary to their natural direction), that is from the eye back to the lightsources. For each pixel of the image, a ray is shot through it from the eye, as illustrated in figure 9.1. The problem is the computation of its color (intensity). First we have to find the first surface point intersected by the ray. If no object is intersected, then the pixel will either take the color of the background, the color of the ambient light or else it will be black. If a surface point is found, then its color has to be calculated. This usually means the calculation of the intensity components at the three representative wavelengths (R, G, B), that is, the *illumination equation* is evaluated in order to obtain the intensity values. The intensity corresponding to a wavelength is composed of ambient, diffuse, specular, coherent reflective and coherent refractive components. For calculating the diffuse and specular

components, a ray is sent towards each lightsource (denoted by s in figure 9.1). If the ray does not hit any object before reaching the lightsource, then the lightsource illuminates the surface point, and the reflected intensity is computed, otherwise the surface point is in shadow with respect to that lightsource. The rays emanated from the surface point towards the lightsources are really called **shadow rays**. For calculating coherent reflective and refractive components, two rays are sent towards the inverse direction of reflection and refraction, respectively (denoted by r and t in figure 9.1). The problem of computing the color of these child rays is the same as for the main ray corresponding to the pixel, so we calculate them recursively:

```
for each pixel p do
    r = ray from the eye through p;
    color of p = Trace(r, 0);
endfor
```

The subroutine **Trace**(r, d) computes the color of the ray r (a dth order reflective or refractive ray) by recursively tracing its reflective and refractive child rays:

```
Trace(r, d)
        if d > dmax then return background color; endif
        q = Intersect(r);                    // q: object surface point
        if q = null then
          return background color;
        endif
        c = AccLightSource(q);                         // c: color
        if object (q) is reflective (coherently) then
          rr = ray towards inverse direction of reflection;
          c += Trace(rr, d + 1);
        endif
        if object (q) is refractive (coherently) then
          rt = ray towards inverse direction of refraction;
          c += Trace(rt, d + 1);
        endif
        return c;
    end
```

The conditional return at the beginning of the routine is needed in order to avoid infinite recurrence (due to total reflection, for example, in the interior of a glass ball). The parameter $d_{\max}$ represents the necessary "depth" limit. It also prevents the calculation of too "distant" generations of rays, since they usually hardly contribute to the color of the pixel due to attenuation at object surfaces. The function **Intersect**(r) gives the intersection point between the ray r and the surface closest to the origin of r if it finds it, and *null* otherwise. The function **AccLightSource**(q) computes the accumulated light intensities coming from the individual lightsources and reaching the surface point q. Usually it is also based on function **Intersect**(r), just like **Trace**(r):

> **AccLightSource**(q)
>> c = ambient intensity + own emission;   // *c: color*
>> **for** each lightsource l **do**
>>> r = ray from q towards l;
>>> **if Intersect**$(r) = $ *null* **then**
>>>> c += diffuse intensity;
>>>> c += specular intensity;
>>> **endif**
>> **endfor**
>> **return** c;
> **end**

The above routine does not consider the illumination of the surface point if the light coming from a lightsource goes through one or more transparent objects. Such situations can be approximated in the following way. If the ray r in the above routine intersects only transparent objects with transmission coefficients $k_t^{(1)}, k_t^{(2)}, \ldots, k_t^{(N)}$ along its path, then the diffuse and specular components are calculated using a lightsource intensity of $k_t^{(1)} \cdot k_t^{(2)} \cdot \ldots \cdot k_t^{(N)} \cdot I$ instead of I, where I is the intensity of the lightsource considered. This is yet another simplification, because refraction on the surface of the transparent objects is ignored here.

It can be seen that the function **Intersect**(r) is the key to recursive ray tracing. Practical observations show that 75–95% of calculation time

is spent on intersection calculations during ray tracing. A **brute force** approach would take each object one by one in order to check for possible intersection and keep the one with the intersection point closest to the origin of r. The calculation time would be proportional to the number of objects in this case. Note furthermore that the function **Intersect**(r) is the only step in ray tracing where the complexity of the calculation is inferred from the number of objects. Hence optimizing the time complexity of the intersection calculation would optimize the time complexity of ray tracing — at least with respect to the number of objects.

9.2   Acceleration of intersection calculations

Let us use the notation $Q(n)$ for the time complexity ("query time") of the routine **Intersect**(r), where n is the number of objects. The brute force approach, which tests each object one by one, requires a query time proportional to n, that is $Q(n) = O(n)$. It is not necessary, however, to test each object for each ray. An object lying "behind" the origin of the ray, for example, will definitely not be intersected by it. But in order to be able to exploit such situations for saving computation for the queries, we must have in store some preliminary information about the spatial relations of objects, because if we do not have such information in advance, all the objects will have to be checked — we can never know whether the closest one intersected is the one that we have not yet checked. The required preprocessing will need computation, and its time complexity, say $P(n)$, will appear. The question is whether $Q(n)$ can be reduced without having to pay too much in $P(n)$.

Working on intuition, we can presume that the best achievable (worst-case) time complexity of the ray query is $Q(n) = O(\log n)$, as it is demonstrated by the following argument. The query can give us $n + 1$ "combinatorially" different answers: the ray either intersects one of the n objects or does not intersect any of them. Let us consider a weaker version of our original query: we do not have to calculate the intersection point exactly, but we only have to report the index of the intersected object (calculating the intersection would require only a constant amount of time if this index

is known). A computer program can be regarded as a numbered list of instructions. The computation for a given input can be characterized by the sequence $i_1, i_2, \ldots, i_m$ of numbers corresponding to the instructions that the program executed, where m is the total number of steps. An instruction can be of one of two types: it either takes the form $X \leftarrow f(X)$, where X is the set of variables and f is an algebraic function, or else it takes the form "IF $f(X) \circ 0$ THEN GOTO i_{yes} ELSE GOTO i_{no}", where $\circ$ is one of the binary relations $=, <, >, \leq, \geq$. The first is called a *decision* instruction; the former is called a *calculational* instruction. Computational instructions do not affect the sequence $i_1, i_2, \ldots, i_m$ directly, that is, if i_j is a calculational instruction, then i_{j+1} is always the same. The sequence is changed directly by the decision instructions: the next one is either i_{yes} or i_{no}. Thus, the computation can be characterized by the sequence $i_1^D, i_2^D, \ldots, i_d^D$ of decision instructions, where d is the number of decisions executed. Since there are two possible further instructions (i_{yes} and i_{no}) for each decision, all the possible sequences can be represented by a binary tree, the root of which represents the first decision instruction, the internal nodes represent intermediate decisions and the leaves correspond to terminations. This model is known as the **algebraic decision tree model** of computation. Since different leaves correspond to different answers, and there are $n + 1$ of them, the length d_{max} of the longest path from the root to any leaf cannot be smaller than the depth of a balanced binary tree with $n + 1$ leaves, that is $d_{\text{max}} = \Omega(\log n)$.

The problem of intersection has been studied within the framework of computational geometry, a field of mathematics. It is called the **ray shooting problem** by computational geometers and is formulated as "given n objects in 3D-space, with preprocessing allowed, report the closest object intersected by any given query ray". Mark de Berg [dB92] has recently developed efficient ray shooting algorithms. He considered the problem for different types of objects (arbitrary and axis parallel polyhedra, triangles with angles greater than some given value, etc.) and different types of rays (rays with fixed origin or direction, arbitrary rays). His most general algorithm can shoot arbitrary rays into a set of arbitrary polyhedra with n edges altogether, with a query time of $O(\log n)$ and preprocessing time and storage of $O(n^{4+\varepsilon})$, where ε is a positive constant that can be made as small as desired. The question of whether the preprocessing and storage complexity are optimal is an open problem. Unfortunately, the complexity

of the preprocessing and storage makes the algorithm not too attractive for practical use.

There are a number of techniques, however, developed for accelerating intersection queries which are suitable for practical use. We can consider them as **heuristic methods** for two reasons. The first is that their approach is not based on complexity considerations, that is, the goal is not a worst-case optimization, but rather to achieve a speed-up for the majority of situations. The second reason is that these algorithms really do not reduce the query time for the worst case, that is $Q(n) = O(n)$. The achievement is that average-case analyses show that they are better than that. We will overview a few of them in the following subsections.

9.2.1   Regular partitioning of object space

Object coherence implies that if a spatial point p is contained by a given object (objects), then other spatial points close enough to p are probably contained by the same object(s). On the other hand, the number of objects intersecting a neighborhood δp of p is small compared with the total number of objects, if the volume of δp is small enough. It gives the following idea for accelerating ray queries. Partition the object space into disjoint cells $C_1, C_2, \ldots, C_m$, and make a list L_i for each cell C_i containing references to objects having non-empty intersection with the cell. If a ray is to be tested, then the cells along its path must be scanned in order until an intersection with an object is found:

> **Intersect**(r)
>   **for** each cell C_k along r in order **do**
>   **if** r intersects at least one object on list L_k **then**
> q = the closest intersection point;
>   **return** q;
>   **endif**
>   **endfor**
>   **return** *null*;
> **end**

Perhaps the simplest realization of this idea is that the set of cells, $C_1, C_2, \ldots, C_m$, consists of congruent axis parallel cubes, forming a regular
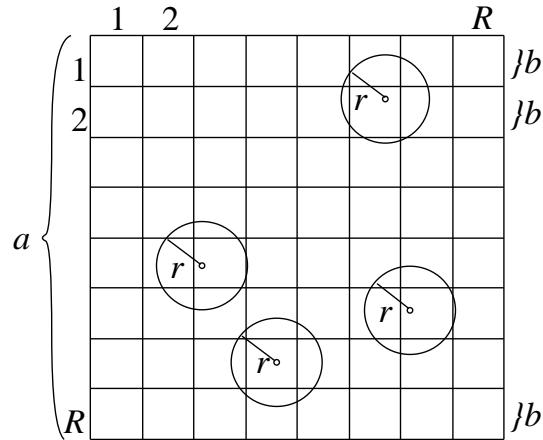
Figure 9.2: Regular partitioning of object space

spatial grid. The outer cycle of the above routine can then be implemented by an incremental line drawing algorithm; Fujimoto *et al.* [FTK86], for instance, used a 3D version of DDA (digital differential analyzer) for this task. If the resolution of the grid is the same, say R, in each of the three spatial directions, then $m = R^3$. The number of cells, k, intersected by a given ray is bounded by:

$$k \leq 1 + 7(R - 1) \tag{9.1}$$

where equality holds for a ray going diagonally (from one corner to the opposite one) through the "big cube", which is the union of the small cells. Thus:

$$k = O(R) = O(\sqrt[3]{m}). \tag{9.2}$$

If we set $m = O(n)$, where n is the number of objects, and the objects are so nicely distributed that the length of the lists L_i remains under a constant value ($|L_i| = O(1)$), then the query time $Q(n)$ can be as low as $O(\sqrt[3]{n})$. In fact, if we allow the objects to be only spheres with a fixed radius r, and assume that their centers are uniformly distributed in the interior of a cube of width a, then we can prove that the expected complexity of the query time can be reduced to the above value by choosing the resolution R properly, as will be shown by the following **stochastic analysis**. One more

assumption will obviously be needed: r must be small compared with a. It will be considered by examining the limiting case $a \to \infty$ with r fixed and n proportional to a^3. The reason for choosing spheres as the objects is that spheres are relatively easy to handle mathematically.
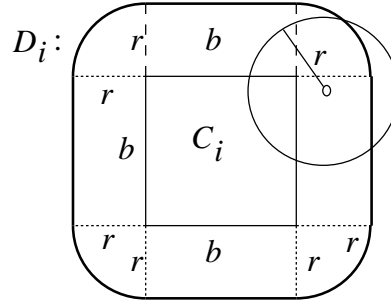
Figure 9.3: Center of spheres intersecting a cell

If points $p_1, \ldots, p_n$ are independently and **uniformly distributed** in the interior of a set X, then the probability of the event that $p_i \in Y \subseteq X$ is:

$$\Pr\{p_i \in Y\} = \frac{|Y|}{|X|} \tag{9.3}$$

where $|\cdot|$ denotes volume. Let X be a cube of width a, and the resolution of the grid of cells be R in all three spatial directions. The cells $C_1, C_2, \ldots, C_m$ will be congruent cubes of width $b = a/R$ and their number is $m = R^3$, as shown in figure 9.2. A sphere will appear on the list L_i corresponding to cell C_i if it intersects the cell. The condition of this is that the center of the sphere falls into a rounded cube shaped region D_i around the cell C_i, as shown in figure 9.3. Its volume is:

$$|D_i| = b^3 + 6b^2r + 3br^2\pi + \frac{4r^3\pi}{3}. \tag{9.4}$$

The probability of the event that a list L_i will contain exactly k elements — exploiting the assumption of uniform distribution — is:

$$
\begin{aligned}
\Pr\{|L_i| = k\} &= \binom{n}{k} \Pr\{p_1, \ldots, p_k \in D_i \ \wedge \ p_{k+1}, \ldots, p_n \notin D_i\} \\
&= \binom{n}{k} \left(\frac{|D_i \cap X|}{|X|}\right)^k \left(\frac{|X \setminus D_i|}{|X|}\right)^{n-k}.
\end{aligned}
\tag{9.5}
$$

If D_i is completely contained by X, then:

$$\Pr\{|L_i| = k\} = \binom{n}{k}\left(\frac{|D_i|}{|X|}\right)^k\left(1 - \frac{|D_i|}{|X|}\right)^{n-k}. \qquad (9.6)$$

Let us consider the limiting behavior of this probability for $n \to \infty$ by setting $a \to \infty$ ($|X| \to \infty$) and forcing $n/|X| \to \rho$, where ρ is a positive real number characterizing the density of the spheres. Note that our uniform distribution has been extended to a **Poisson point process** of intensity ρ. Taking the above limits into consideration, one can derive the following limiting behavior of the desired probability:

$$\Pr'\{|L_i| = k\} = \lim_{\substack{|X| \to \infty \\ n/|X| \to \rho}} \Pr\{|L_i| = k\} = \frac{(\rho|D_i|)^k}{k!}e^{-\rho|D_i|}. \qquad (9.7)$$

Note that the rightmost expression characterizes a Poisson distribution with parameter $\rho|D_i|$, as the limit value of the binomial distribution on the right-hand side of expression 9.6 for $n \to \infty$ and $n/|X| \to \rho$. The expected length of list L_i is then given by the following formula:

$$E[|L_i|] = \sum_{k=1}^{\infty} k \cdot \Pr'\{|L_i| = k\} = \rho|D_i|. \qquad (9.8)$$

Substituting expression 9.4 of the volume $|D_i|$, and bearing in mind that $n/|X| \to \rho$ and $|X| = a^3 = R^3 b^3$ hence $b^3 \to n/\rho R^3$, we can get:

$$E[|L_i|] = \frac{n}{R^3} + 6\rho^{1/3}r\frac{n^{2/3}}{R^2} + 3\rho^{2/3}r^2\pi\frac{n^{1/3}}{R} + \rho\frac{4r^3\pi}{3} \quad (1 \le i \le R^3). \quad (9.9)$$

for the expected asymptotic behavior of the list length. This quantity can be kept independent of n (it can be $O(1)$) if is R chosen properly. The last term tends to be constant, independently of R. The first term of the sum requires $R^3 = \Omega(n)$, at least. The two middle terms will also converge to a constant with this choice, since then $R^2 = \Omega(n^{2/3})$ and $R = \Omega(n^{1/3})$. The conclusion is the following: if our object space X is partitioned into congruent cubes with an equal resolution R along all three spatial directions, and R is kept $R = \Omega(\sqrt[3]{n})$, then the expected number of spheres intersecting any of the cells will be $O(1)$, independent of n in the asymptotic sense. This implies furthermore (cf. expression 9.1) that the number of cells along the

path of an arbitrary ray is also bounded by $O(\sqrt[3]{n})$. The actual choice for R can modify the constant factor hidden by the "big O", but the last term of the sum does not allow us to make it arbitrarily small. The value $R = \lceil \sqrt[3]{n} \rceil$ seems to be appropriate in practice ($\lceil \cdot \rceil$ denotes "ceiling", that is the smallest integer above or equal). We can conclude that the expected query time and expected storage requirements of the method are:

$$E[Q(n)] = O(R(n)) = O(\sqrt[3]{n}) \quad \text{and} \quad E[S(n)] = O(n) \tag{9.10}$$

respectively, for the examined distribution of sphere centers. The behavior
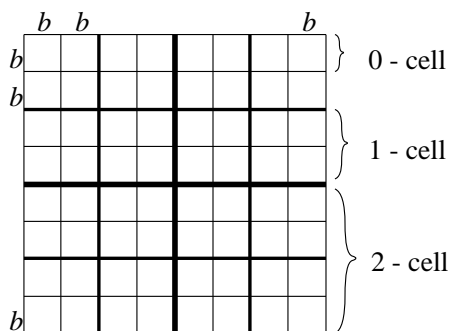
Figure 9.4: j-cells

of the preprocessing time $P(n)$ depends on the efficiency of the algorithm used for finding the intersecting objects (spheres) for the individual cells. Let us consider the 8 neighboring cells of width b around their common vertex. Their union is a cube of width $2b$. An object can intersect any of the 8 cells only if it intersects the cube of width $2b$. Furthermore, considering the union of 8 such cubes, which is a cube of width $4b$, a similar statement can be made, etc. In order to exploit this idea, let us choose $R = 2^K$ with $K = \lceil (\log_2 n)/3 \rceil$, in order to satisfy the requirement $R = \Omega(\sqrt[3]{n})$. The term j-cell will be used to denote the cubes of width $2^j b$ containing 2^{3j} cells of width b, as shown in figure 9.4. Thus, the smallest cells C_i become 0-cells, denoted by $C_i^{(0)}$ ($1 \leq i \leq 2^{3K}$), and the object space X itself will appear as the sole K-cell. The preprocessing algorithm will progressively refine the partitioning of the object space, which will consist of one K-cell in the first

step, 8 $(K-1)$-cells in the second step, and $2^{3K} = O(n)$ 0-cells in the last step.

The algorithm is best shown as a recursive algorithm, which preprocesses a list $L^{(j)}$ of objects with respect to a j-cell $C_i^{(j)}$. Provided that the object scene containing the objects $o_1, \ldots, o_n$ is enclosed by a cube (or rectangular box) X, it can be preprocessed by invoking a subroutine call **Preprocess**$(\{o_1, \ldots, o_n\}^{(K)}, X^{(K)})$ (with $K = \lceil (\log_2 n)/3 \rceil$), where the subroutine **Preprocess** is the following:

> **Preprocess**$(L^{(j)}, C_i^{(j)})$
>> **if** $j = 0$ **then** $L_i = L^{(j)}$; **return** ;
>> **for** each subcell $C_k^{(j-1)}$ $(1 \le k \le 8)$ contained by $C_i^{(j)}$ **do**
>>> $L^{(j-1)} = \{\}$;
>>> **for** each object o on list $L^{(j)}$ **do**
>>>> **if** o intersects $C_k^{(j-1)}$ **then**
>>>>> add o to $L^{(j-1)}$;
>>>> **endif**
>>> **endfor**
>>> **Preprocess**$(L^{(j-1)}, C_k^{(j-1)})$;
>> **endfor**
> **end**

The algorithm can be speeded up by using the trick that if the input list corresponding to a j-cell becomes empty ($|L^{(j)}| = 0$) at some stage, then we do not process the "child" cells further but return instead. The maximal depth of recurrence is K, because j is decremented by 1 at each recursive call, hence we can distinguish between $K+1$ different levels of execution. Let the level of executing the uppermost call be K, and generally, the level of execution be j if the superscript $^{(j)}$ appears in the input arguments. The execution time $T = P(n)$ of the preprocessing can be taken as the sum $T = T_0 + T_1 + \ldots + T_K$, where the time T_j is spent at level j of execution. The routine is executed only once at level K, 8 times at level $K-1$, and generally:

$$N_j = 2^{3(K-j)} \quad K \ge j \ge 0 \tag{9.11}$$

times at level j. The time taken for a given instance of execution at level j is proportional to the actual length of the list $L^{(j)}$ to be processed. Its

expected length is equal to the expected number of objects intersecting the corresponding j-cell $C_i^{(j)}$. Its value is:

$$E[|L^{(j)}|] = \rho|D_i^{(j)}| \tag{9.12}$$

where $D_i^{(j)}$ is the rounded cube shaped region around the j-cell $C_i^{(j)}$, very similar to that shown in figure 9.3, with the difference that the side of the "base cube" is $2^{j-K}a$. Its volume is given by the formula:

$$|D_i^{(j)}| = 2^{3(j-K)}a^3 + 6 \cdot 2^{2(j-K)}a^2r + 3 \cdot 2^{j-K}ar^2\pi + \frac{4r^3\pi}{3} \tag{9.13}$$

which is the same for each j-cell. Thus, the total time T_j spent at level j of execution is proportional to:

$$N_j\rho|D_i^{(j)}| = \rho a^3 + 6 \cdot 2^{K-j}\rho a^2r + 3 \cdot 2^{2(K-j)}\rho ar^2\pi + 2^{3(K-j)}\rho\frac{4r^3\pi}{3}. \tag{9.14}$$

Let us sum these values for $1 \leq j \leq K - 1$, taking the following identity into consideration:

$$2^i + 2^{i\cdot 2} + \ldots + 2^{i\cdot(K-1)} = \frac{2^{i\cdot K} - 2^i}{2^i - 1} \quad (i \geq 1), \tag{9.15}$$

where i refers to the position of the terms on the right-hand side of expression 9.14 ($i = 1$ for the second term, $i = 2$ for the third etc.). Thus the value $T_1 + \ldots + T_{K-1}$ is proportional to:

$$(K - 1)\rho a^3 + 6 \cdot \frac{2^K - 2}{1}\rho a^2r + 3 \cdot \frac{2^{2K} - 4}{3}\rho ar^2\pi + \frac{2^{3K} - 8}{7}\rho\frac{4r^3\pi}{3}. \tag{9.16}$$

Since $K = O(\log n)$ and $a^3 \to n/\rho$, the first term is in the order of $O(n \log n)$, and since $n^{i/3} \leq 2^{iK} < 2n^{i/3}$, the rest of the terms are only of $O(n)$ (actually, this is in connection with the fact that the center of the majority of the spheres intersecting a cube lies also in the cube as the width of the cube increases). Finally, it can easily be seen that the times T_0 and T_K are both proportional to n, hence the expected preprocessing time of the method is:

$$E[P(n)] = O(n \log n) \tag{9.17}$$

for the examined Poisson distribution of sphere centers.

We intended only to demonstrate here how a stochastic average case analysis can be performed. Although the algorithm examined here is relatively simple compared to those coming in the following subsections, performing the analysis was rather complicated. This is the reason why we will not undertake such analyses for the other algorithms (they are to appear in [Már94]).

9.2.2   Adaptive partitioning of object space

The regular cell grid is very attractive for the task of object space subdivision, because it is simple, and the problem of enumerating the cells along the path of a ray is easy to implement by means of a 3D incremental line generator. The cells are of the same size, wherever they are. Note that we are solely interested in finding the intersection point between a ray and the *surface* of the closest object. The number of cells falling totally into the interior of an object (or outside all the objects) can be very large, but the individual cells do not yield that much information: each of them tells us that there is no ray-surface intersection inside. Thus, the union of such cells carries the same information as any of them do individually — it is not worth storing them separately. The notion and techniques used in the previous subsection form a good basis for showing how this idea can be exploited.
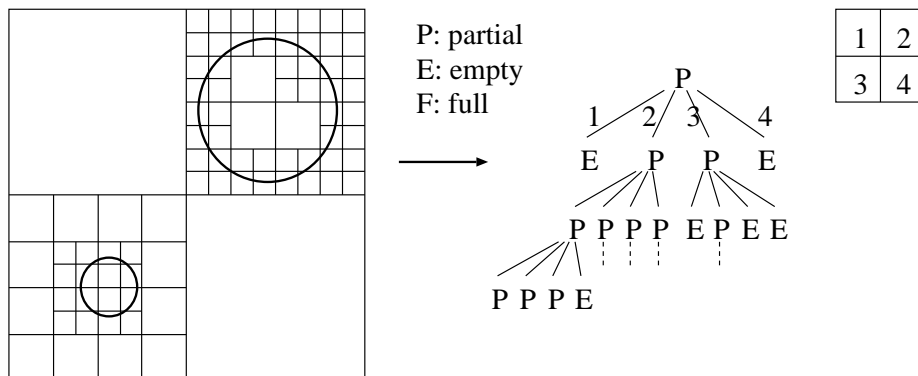
Figure 9.5: The octree structure for space partitioning

If our object space is enclosed by a cube of width a, then the resolution of subdivision, R, means that the object space was subdivided into congruent cubes of width $b = a/R$ in the previous subsection. We should remind the reader that a cube of width $2^j b$ is called a j-cell, and that a j-cell is the union of exactly 2^{3j} 0-cells. Let us distinguish between three types of j-cell: an *empty* cell has no intersection with any object, a *full* cell is completely contained in one or more objects, and a *partial* cell contains a part of the surface of at least one object. If a j-cell is empty or full, then we do not have to divide it further into $(j - 1)$-cells, because the child cells would also be empty or full, respectively. We subdivide only partial cells. Such an uneven subdivision can be represented by an **octree** (octal tree) structure, each node of which has either 8 or no children. The two-dimensional analogue of the octree (the quadtree) is shown in figure 9.5. A node corresponds to a j-cell in general, and has 8 children $((j - 1)$-cells) if the j-cell is partial, or has no children if it is empty or full. If we use it for ray-surface intersection calculations, then only partial cells need have references to objects, and only to those objects whose surface intersects the cell.

The preprocessing routine that builds this structure is similar to the one shown in the previous subsection but with the above mentioned differences. If the objects of the scene X are $o_1, \ldots, o_n$, then the forthcoming algorithm must be invoked in the following form: **Preprocess**$(\{o_1, \ldots, o_n\}^{(K)}, X^{(K)})$, where K denotes the allowed number of subdivision steps at the current recurrence level. The initial value $K = \lceil (\log_2 n)/3 \rceil$ is proper again, since our subdivision can become a regular grid in the worst case. The algorithm will return the octree structure corresponding to X. The notation $L(C_i^{(j)})$ in the algorithm stands for the object reference list corresponding to the j-cell $C_i^{(j)}$ (if it is partial), while $R_k(C_i^{(j)})$ $(1 \leq k \leq 8)$ stands for the reference to its kth child (*null* denotes no child).

The algorithm is then the following:

> **Preprocess**$(L^{(j)}, C_i^{(j)})$
> > **if** $j = 0$ **then**   // *bottom of recurrence*
> > > $R_1(C_i^{(j)}) = \ldots = R_8(C_i^{(j)}) = null;$
> > > $L(C_i^{(j)}) = L^{(j)};$ **return** $C_i^{(j)};$
> >
> > **endif**
> > **for** each subcell $C_k^{(j-1)}$ $(1 \le k \le 8)$ contained by $C_i^{(j)}$ **do**
> > > $L^{(j-1)} = \{\};$
> > > **for** each object o on list $L^{(j)}$ **do**
> > > > **if** surface of o intersects $C_k^{(j-1)}$ **then** add o to $L^{(j-1)};$
> > >
> > > **endfor**
> > > **if** $L^{(j-1)} = \{\}$ **then**   // *empty or full*
> > > > $R_k(C_i^{(j)}) = null;$
> > >
> > > **else**   // *partial*
> > > > $R_k(C_i^{(j)}) = $ **Preprocess**$(L^{(j-1)}, C_k^{(j-1)});$
> > >
> > > **endif**
> >
> > **endfor**
> > **return** $C_i^{(j)};$
>
> **end**

The method saves storage by its adaptive operation, but raises a new problem, namely the enumeration of cells along the path of a ray during ray tracing.

The problem of visiting all the cells along the path of a ray is known as **voxel walking** (voxel stands for "volume cell" such as pixel is "picture cell"). The solution is almost facile if the subdivision is a regular grid, but what can we do with our octree? The method commonly used in practice is based on a **generate-and-test** approach, originally proposed by Glassner [Gla84]. The first cell the ray visits is the cell containing the origin of the ray. In general, if a point p is given, then the cell containing it can be found by recursively traversing the octree structure from its root down to the leaf containing the point.

This is what the following routine does:

> **Classify**$(p, C_i^{(j)})$
>   **if** $C_i^{(j)}$ is a leaf $(R_k(C_i^{(j)})=null)$ **then return** $C_i^{(j)}$;
>   **for** each child $R_k(C_i^{(j)})$ $(1 \le k \le 8)$ **do**
>   **if** subcell $R_k(C_i^{(j)})$ contains p **then**
>   **return Classify**$(p, R_k(C_i^{(j)}))$;
>   **endif**
>   **endfor**
>   **return** *null*;
> **end**

The result of the function call **Classify**$(p, X^{(K)}))$ is the cell containing a point $p \in X$. It is *null* if p falls outside the object space X. The worst case time required by the classification of a point will be proportional to the depth of the octree, which is $K = \lceil(\log_2 n)/3\rceil$, as suggested earlier. Once the cell containing the origin of the ray is known, the next cell visited can be determined by first generating a point q which definitely falls in the interior of the next cell, and then by testing to find which cell contains q. Thus, the intersection algorithm will appear as follows (the problem of generating a point falling into the next cell will be solved afterwards):

> **Intersect**(r)
> $o_{\min} = null$;   // $o_{\min}$: *closest intersected object*
> p = origin of ray;
> $C = $ **Classify**$(p, X^{(K)})$;
>   **while** $C \ne null$ **do**
>   **for** each object o on list $L(C)$ **do**
>   **if** r intersects o closer than $o_{\min}$ **then** $o_{\min} = o$;
>   **endfor**
>   **if** $o_{\min} \ne null$ **then return** $o_{\min}$;
> q = a point falling into the next cell;
> $C = $ **Classify**$(q, X^{(K)})$;
>   **endwhile**
>   **return** *null*;
> **end**

There is only one step to work out, namely how to generate a point q which falls definitely into the neighbor cell. The point where the ray r exits the actual cell can easily be found by intersecting it with the six faces of the cell. Without loss of generality, we can assume that the direction vector of r has nonnegative x, y and z components, and its exit point e either falls in the interior of a face with a normal vector incident with the x coordinate axis, or is located on an edge of direction incident with the z axis, or is a vertex of the cell — all the other combinations can be handled similarly. A proper q can be calculated by the following vector sums, where $\vec{x}, \vec{y}$ and $\vec{z}$ represent the (unit) direction vectors of the coordinate axes, and $b = a2^{-K}$ is the side of the smallest possible cell, and the subscripts of q distinguish between the three above cases in order:

$$q_1 = e + \frac{b}{2}\vec{x}, \quad q_2 = e + \frac{b}{2}\vec{x} + \frac{b}{2}\vec{y} \quad \text{and} \quad q_3 = e + \frac{b}{2}\vec{x} + \frac{b}{2}\vec{y} + \frac{b}{2}\vec{z}. \quad (9.18)$$

For the suggested value of the subdivision parameter K, the expected query time will be $\mathrm{E}[Q(n)] = O(\sqrt[3]{n}\log n)$ per ray if we take into consideration that the maximum number of cells a ray intersects is proportional to $R = 2^K$ (cf. expression 9.1), and the maximum time we need for stepping into the next cell is proportional to K.

9.2.3 Partitioning of ray space

A ray can be represented by five coordinates, $x, y, z, \vartheta, \varphi$ for instance, the first three of which give the origin of the ray in the 3D space, and the last two define the direction vector of the ray as a point on the 2D surface of the unit sphere (in polar coordinates). Thus, we can say that a ray r can be considered as a point of the 5D ray-space $\Re^5 = E^3 \times O^2$, where the first space is a Euclidean space, the second is a spherical one, and their Cartesian product is a cylinder-like space. If our object space, on the other hand, contains the objects $o_1, \ldots, o_n$, then for each point (ray) $r \in \Re^5$, there is exactly one $i(r) \in \{0, 1, \ldots, n\}$ assigned, where $i(r) = 0$ if r intersects no object, and $i(r) = j$ if r intersects object o_j first. We can notice furthermore that the set of rays intersecting a given object o_j — that is the regions $R(j) = \{r \mid i(r) = j\}$ — form connected subsets of $\Re^5$, and $R(0) \cup R(1) \cup \ldots \cup R(n) = \Re^5$, that is, the $n + 1$ regions form a subdivision of the ray space. This leads us to hope that we can construct a ray-object

intersection algorithm with a good (theoretically optimal $O(\log n)$) query time based on the following **locus approach**: first we build the above mentioned subdivision of the ray space in a preprocessing step, and then, whenever a ray r is to be tested, we *classify* it into one of the $n + 1$ regions, and if the region containing r is $R(j)$, then the intersection point will be on the surface of o_j. The only problem is that this subdivision is so difficult to calculate that nobody has even tried it yet. Approximations, however, can be carried out, which Arvo and Kirk in fact did [AK87] when they worked out their method called **ray classification**. We shall outline their main ideas here.
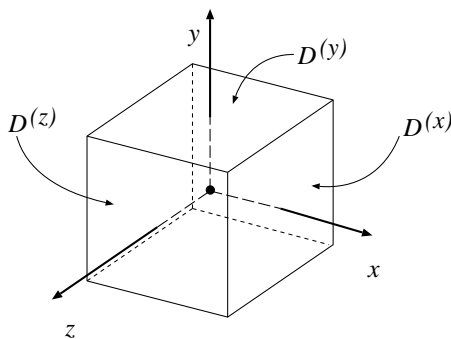
Figure 9.6: The direction cube

A crucial problem is that the ray space $\Re^5$ is not Euclidean (but cylinder-like), hence it is rather difficult to treat computationally. It is very inconvenient namely, that the points representing the direction of the rays are located on the surface of a sphere. We can, however, use a more suitable representation of direction vectors which is not "curved". Directions will be represented as points on the surface of the unit cube, instead of the unit sphere, as shown in figure 9.6. There are discontinuities along the edges of the cube, so the direction space will be considered as a collection $D^{(x)}, D^{(-x)}, D^{(y)}, D^{(-y)}, D^{(z)}, D^{(-z)}$ of six spaces (faces of the unit cube), each containing the directions with the main component (the one with the greatest absolute value) being in the same coordinate direction $(x, -x, y, -y, z, -z)$. If the object scene can be enclosed by a cube E — containing the eye as well — then any ray occurring during ray tracing must

fall within one of the sets in the collection:

$$H = \{E \times D^{(x)}, E \times D^{(-x)}, E \times D^{(y)}, E \times D^{(-y)}, E \times D^{(z)}, E \times D^{(-z)}\}. \quad (9.19)$$

Each of the above six sets is a 5D hypercube. Let us refer to this collection H as the bounding hyperset of our object scene.
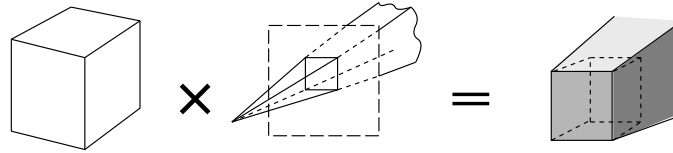
Figure 9.7: Beams of rays in 3D space

The hyperset H will be recursively subdivided into cells $H^{(1)}$, ..., $H^{(m)}$ (each being an axis parallel hypercube), and a *candidate list* $L(H^{(i)})$ will be associated with each cell $H^{(i)}$ containing references to objects that are intersected by any ray $r \in H^{(i)}$. Each such hypercube $H^{(i)}$ is a collection of rays with their origin in a 3D rectangular box and their direction falling into an axis parallel 2D rectangle embedded in the 3D space. These rays form an unbounded polyhedral volume in the 3D space, called a **beam**, as shown in figure 9.7. An object appears on the list associated with the 5D hypercube if and only if it intersects the 3D beam corresponding to the hypercube. At each step of subdivision a cell will be divided into two halves along one of the five directions. If we normalize the object scene so that the enclosing cube E becomes a unit cube, then we can decide to subdivide a 5D cell along one of its longest edges. Such a subdivision can be represented by a binary tree, the root of which corresponds to H itself, the two children correspond to the two halves of H, etc. In order to save computation, the subdivision will not be built completely by a separate preprocessing step, but rather the hierarchy will be constructed adaptively during ray tracing by **lazy evaluation**. Arvo and Kirk suggested [AK87] terminating this subdivision when either the candidate list or the hypercube fall below a fixed size threshold. The heuristic reasoning is that "a small candidate set indicates that we have achieved the goal of making the associated rays inexpensive to intersect with the environment", while "the hypercube size constraint is imposed to allow the cost of creating a candidate set to be

amortized over many rays" (cited from [AK87]). The intersection algorithm then appears as follows, where $R_l(H')$ and $R_r(H')$ denote the left and right children of cell H' in the tree structure, n_{min} is the number under which the length of an object list is considered to be as "small enough", and w_{min} denotes the minimal width of a cell (width of cells is taken as the smallest width along the five axes).

> **Intersect**(r)
> $H' = $ **Classify**(r, H);
>   **while** $|L(H')| > n_{min}$ and $|H'| > w_{min}$ **do**
> $H'_l, H'_r = $ two halves of H'; $L(H'_l) = \{\}$; $L(H'_r) = \{\}$;
>   **for** each object o on list $L(H')$ **do**
>   **if** o intersects the beam of H'_l **then** add o to $L(H'_l)$;
>   **if** o intersects the beam of H'_r **then** add o to $L(H'_r)$;
>   **endfor**
> $R_l(H') = H'_l$; $R_r(H') = H'_r$; $H' = $ **Classify**(r, H');
>   **endwhile**
> $o_{min} = null$;   // o_{min}: *closest intersected object*
>   **for** each object o on list $L(H')$ **do**
>   **if** r intersects o closer than o_{min} **then** $o_{min} = o$; **endif**
>   **endfor**
>   **return** o_{min};
> **end**

The routine **Classify**(r, H') called from the algorithm finds the smallest 5D hypercube containing the ray r by performing a binary search in the tree with root at H'.

9.2.4   Ray coherence theorems

Two rays with the same origin and slightly differing directions probably intersect the same object, or more generally, if two rays are close to each other in the 5D ray space then they probably intersect the same object. This is yet another guise of object coherence, and we refer to it as **ray coherence**. Closeness here means that both the origins and the directions are close. The ray classification method described in the previous section used a 5D subdivision along all the five ray parameters, the first three of

which represented the origin of the ray in the 3D object space, hence every ray originating in the object scene is contained in the structure, even those that have their origins neither on the surface of an object nor in the eye position. These rays will definitely not occur during ray tracing. We will define *equivalence classes* of rays in an alternative way: two rays will be considered to be equivalent if their origins are on the surface of the same object and their directions fall in the same class of a given partition of the direction space. This is the main idea behind the method of Ohta and Maekawa [OM87]. We will describe it here in some detail.

Let the object scene consist of n objects, including those that we would like to render, the lightsources and the eye. Some, say m, of these n objects are considered to be *ray origins*, these are the eye, the lightsources and the reflective/refractive objects. The direction space is partitioned into d number of classes. This subdivision can be performed by subdividing each face of the direction cube (figure 9.6) into small squares at the desired resolution. The preprocessing step will build a two-dimensional array $O[1 \ldots m, 1 \ldots d]$, containing lists of references to objects. An object o_k will appear on the list at $O[i, j]$ if there exists a ray intersecting o_k with its origin on o_i and direction in the jth direction class. Note that the cells of the array O correspond to the "equivalence classes" of rays defined in the previous paragraph. If this array is computed, then the intersection algorithm becomes very simple:

> **Intersect**(r)
> $\quad i =$ index of object where r originates;
> $\quad j =$ index of direction class containing the direction of r;
> $\quad o_{\min} = null;$ $\qquad\qquad\qquad$ // $o_{\min}$: *closest intersected object*
> $\quad$**for** each object o on list $O[i, j]$ **do**
> $\quad\quad$**if** r intersects o closer than $o_{\min}$ **then**
> $\quad\quad\quad o_{\min} = o;$
> $\quad\quad$**endif**
> $\quad$**endfor**
> $\quad$**return** $o_{\min};$
> **end**

The computation of the array O is based on the following geometric considerations. We are given two objects, o_1 and o_2. Let us define a set $V(o_1, o_2)$ of directions, so that V contains a given direction δ if and only if there exists

a ray of direction δ with its origin on o_1 and intersecting o_2, that is:

$$V(o_1, o_2) = \{\delta \mid \exists r : \text{org}(r) \in o_1 \ \wedge \ \text{dir}(r) = \delta \ \wedge \ r \cap o_2 \neq \emptyset\} \qquad (9.20)$$

where $\text{org}(r)$ and $\text{dir}(r)$ denote the origin and direction of ray r, respectively. We will call the set $V(o_1, o_2)$ the **visibility set** of o_2 with respect to o_1 (in this order). If we are able to calculate the visibility set $V(o_i, o_k)$ for a pair of objects o_i and o_k, then we have to add o_k to the list of those cells in the row $O[i, 1 \ldots d]$ of our two-dimensional array which have non-empty intersection with $V(o_i, o_k)$. Thus, the preprocessing algorithm can be the following:

Preprocess$(o_1, \ldots, o_n)$
    initialize each list $O[i, j]$ to $\{\}$;
    for each ray origin o_i $(1 \leq i \leq m)$ **do**
        for each object o_k $(1 \leq k \leq n)$ **do**
            compute the visibility set $V(o_i, o_k)$;
            for each direction class Δ_j with $\Delta_j \cap V(o_i, o_k) \neq \emptyset$ **do**
                add o_k to list $O[i, j]$;
            endfor
        endfor
    endfor
end
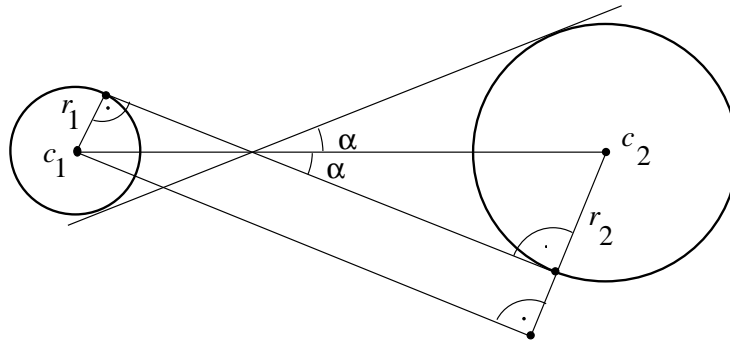
Figure 9.8: Visibility set of two spheres

The problem is that the exact visibility sets can be computed only for a narrow range of objects. These sets are subsets of the surface of the unit
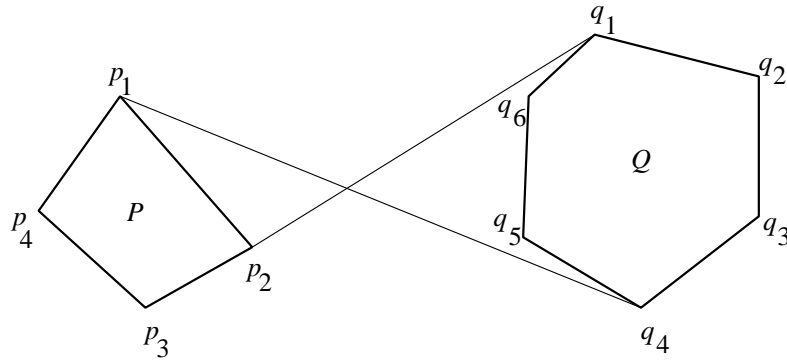
Figure 9.9: Visibility set of two convex hulls

sphere — or alternatively the direction cube. Ohta and Maekawa [OM87] gave the formula for a pair of spheres, and a pair of convex polyhedra. If S_1 and S_2 are spheres with centers c_1, c_2 and radii r_1, r_2, respectively, then $V(S_1, S_2)$ will be a spherical circle. Its center is at the spherical point corresponding to the direction $\vec{c_1 c_2}$ and its (spherical) radius is given by the expression $\arcsin\{(r_1 + r_2)/|c_1 - c_2|\}$, as illustrated in figure 9.8. If P and Q are convex polyhedra with vertices $p_1, \ldots, p_n$ and $q_1, \ldots, q_m$, respectively, then $V(P, Q)$ will be the spherical convex hull of $n \cdot m$ spherical points corresponding to the directions $\vec{p_1 q_1}, \ldots, \vec{p_1 q_m}, \ldots, \vec{p_n q_1}, \ldots, \vec{p_n q_m}$ (see figure 9.9). It can be shown [HMSK92] that for a mixed pair of a convex polyhedron P with vertices $p_1, \ldots, p_n$ and a sphere S with center c and radius r, $V(P, S)$ is the spherical convex hull of n circles with centers at $\vec{p_1 c}, \ldots, \vec{p_n c}$ and radii $\arcsin\{r/|p_1 - c|\}, \ldots, \arcsin\{r/|p_n - c|\}$. In fact, these circles are nothing else than the visibility sets $V(p_1, S), \ldots, V(p_n, S)$, corresponding to the vertices of P. This gives the idea of a generalization of the above results in the following way [MRSK92]: If A and B are convex hulls of the sets $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$, respectively, then $V(A, B)$ will be the spherical convex hull of the visibility sets $V(A_1 B_1), \ldots, V(A_1 B_m), \ldots, V(A_n B_1), \ldots, V(A_n B_m)$. Note that disjointness for the pairs of objects was assumed so far, because if the objects intersect then the visibility set is the whole sphere surface (direction space).

Unfortunately, exact expression of visibility sets is not known for further types of objects. We can use approximations, however. Any object can be

enclosed by a large enough sphere, or a convex polyhedron, or a convex hull of some sets. The simpler the enclosing shell is, the easier the calculations are, but the greater the difference is between the real and the computed visibility set. We always have to find a trade-off between accuracy and computation time.

9.3   Distributed ray tracing

Recursive ray tracing is a very elegant method for simulating phenomena such as shadows, mirror-like reflections, and refractions. The simplifications in the illumination model — point-like lightsources and point-sampling (infinitely narrow light rays) — assumed so far, however, cause sharp shadows, reflections and refractions, although these phenomena usually occur in a **blurred** form in reality.

Perhaps the most elegant method among all the proposed approaches to handle the above mentioned blurred (fuzzy) phenomena is the so-called **distributed ray tracing** due to Cook *et al.* [CPC84]. The main advantage of the method is that phenomena like motion blur, depth of field, penumbras, translucency and fuzzy reflections are handled in an easy and somewhat unified way with no additional computational cost beyond those required by spatially oversampled ray tracing. The basic ideas can be summarized as follows. Ray tracing is a kind of **point sampling** and, as such, is a subject to aliasing artifacts (see chapter 11 on Sampling and Quantization Artifacts). The usual way of reducing these artifacts is the use of some post-filtering technique on an oversampled picture (that is, more image rays are generated than the actual number of pixels).

The key idea is, that oversampling can be made not only in space but also in the time (motion sampling), on the area of the camera lens or the entire shading function. Furthermore, "not everything must be sampled everywhere" but rather the rays can be *distributed*. In the case of motion sampling, for example, instead of taking multiple time samples at every spatial location, the rays are distributed in time so that rays at different spatial locations trace the object scene at different instants of time.

Distributing the rays offers the following benefits at little additional cost:

- Distributing reflected rays according to the specular distribution function produces gloss (fuzzy reflection).

- Distributing transmitted rays produces blurred transparency.

- Distributing shadow rays in the solid angle of the lightsources produces penumbras.

- Distributing rays on the area of the camera lens produces depth of field.

- Distributing rays in time produces motion blur.

Oversampled ray traced images are generated by emanating more than one ray through the individual pixels. The rays corresponding to a given pixel are usually given the same origin (the eye position) and different direction vectors, and because of the different direction vectors, the second and further generation rays will generally have different origins and directions, as well. This spatial oversampling is generalized by the idea of distributing the rays. Let us overview what distributing the rays means in concrete situations.

Fuzzy shading

We have seen in chapter 3 that the intensity I^{out} of the reflected light coming from a surface point towards the viewing position can be expressed by an integral of an illumination function $I^{\text{in}}(\vec{L})$ ($\vec{L}$ is the incidence direction vector) and a reflection function over the hemisphere about the surface point (cf. equation 3.30):

$$I_r^{\text{out}} = k_r \cdot I_r^{\text{in}} + \int\limits^{2\pi} I^{\text{in}}(\vec{L}) \cdot \cos \phi_{\text{in}} \cdot R^*(\vec{L}, \vec{V}) \, d\omega_{\text{in}} \qquad (9.21)$$

where $\vec{V}$ is the viewing direction vector and the integration is taken over all the possible values of $\vec{L}$. The coherent reflection coefficient k_r is in fact a δ-function, that is, its value is non-zero only at the reflective inverse of the viewing direction $\vec{V}$. Sources of second or higher order reflections

are considered only from this single direction (the incoming intensity I_r^{in} is computed recursively). A similar equation can be given for the intensity of the refracted light:

$$I_t^{\text{out}} = k_t \cdot I_t^{\text{in}} + \int\limits^{2\pi} I^{\text{in}}(\vec{L}) \cdot \cos\phi_{\text{in}} \cdot T^*(\vec{L}, \vec{V})\, d\omega_{\text{in}} \qquad (9.22)$$

where the integration is taken over the hemisphere below the surface point (in the interior of the object), I_t^{in} is the intensity of the coherent refractive (transmissive) illumination and k_t is the coherent transmission coefficient (also a δ-function).

The integrals in the above expressions are usually replaced by finite sums according to the finite number of (usually) point-like or directional light-sources. The effects produced by finite extent lightsources can be considered by distributing more than one shadow ray over the solid angle of the visible portion of each lightsource. This technique can produce **penumbras**. Furthermore, second and higher order reflections need no longer be restricted to single directions but rather the reflection coefficient k_r can be treated as non-zero over the whole hemisphere and more than one rays can be distributed according to its function. This can model **gloss**. Finally, distributing the refracted rays in a similar manner can produce **blurred translucency**.

Depth of field

Note that the usual projection technique used in computer graphics in fact realizes a pinhole camera model with each object in sharp focus. It is an idealization, however, of a real camera, where the ratio of the focal length F and the diameter D of the lens is a finite positive number, the so-called **aperture** number a:

$$a = \frac{F}{D}. \qquad (9.23)$$

The finite aperture causes the effect called **depth of field** which means that object points at a given distance appear in sharp focus on the image and other points beyond this distance or closer than that are confused, that is, they are mapped to finite extent patches instead of points.

It is known from geometric optics (see figure 9.10) that if the focal length of a lens is F and an object point is at a distance T from the lens, then
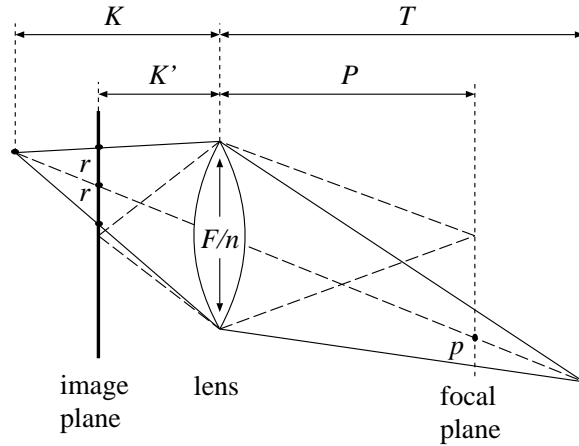
Figure 9.10: Geometry of lens

the corresponding image point will be in sharp focus on an image plane at a distance K behind the lens, where F, T and K satisfy the equation:

$$\frac{1}{F} = \frac{1}{K} + \frac{1}{T}. \tag{9.24}$$

If the image plane is not at the proper distance K behind the lens but at a distance K', as in figure 9.10, then the object point maps onto a circle of radius r:

$$r = \frac{1}{K} |K - K'| \frac{F}{n}. \tag{9.25}$$

This circle is called the **circle of confusion** corresponding to the given object point. It expresses that the color of the object point affects the color of not only a single pixel but all the pixels falling into the circle.

A given camera setting can be specified in the same way as in real life by the aperture number a and the *focal distance*, say P (see figure 9.10), which is the distance of those objects from the lens, which appear in sharp focus on the image (not to be confused with the focal length of the lens). The focal length F is handled as a constant. The plane at distance P from the lens is called the *focal plane*. Both the distance of the image plane from the lens and the diameter (D) of the lens can be calculated from these parameters using equations 9.24 and 9.23, respectively.

In depth of field calculations, the eye position is imagined to be in the center of the lens. First a ray is emanated from the pixel on the image plane through the eye position, as in usual ray tracing, and its color, say I_0 is computed. Let the point where this "traditional" ray intersects the focal plane be denoted by p. Then some further points are selected on the surface of the lens, and a ray is emanated from each of them through point p. Their colors, say $I_1, \ldots, I_m$, are also computed. The color of the pixel will be the average of the intensities $I_0, I_1, \ldots, I_m$. In fact, it approximates an integral over the lens area.

Motion blur

Real cameras have a finite *exposure time*, that is, the film is illuminated during a time interval of nonzero width. If some objects are in motion, then their image will be blurred on the picture, and the higher the speed of an object is, the longer is its trace on the image. Moreover, the trace of an object is translucent, that is, the objects behind it become partially visible. This effect is known as **motion blur**. This is yet another kind of integration, but now in time. Distributing the rays in time can easily be used for approximating (sampling) this integral. It means that the different rays corresponding to a given pixel will correspond to different time instants. The path of motion can be arbitrarily complex, the only requirement is the ability to calculate the position of any object at any time instant.

We have seen that distributed ray tracing is a unified approach to modeling realistic effects such as fuzzy shading, depth of field or motion blur. It approximates the analytic function describing the intensity of the image pixels at a higher level than usual ray tracing algorithms do. Generally this function involves several nested integrals: integrals of illumination functions multiplied by reflection or refraction functions over the reflection or transmission hemisphere, integrals over the surface of the lens, integrals over time, integrals over pixel area. This integral is so complicated that only approximation techniques can be used in practice. Distributing the rays is in fact a point sampling method performed on a multi-dimensional parameter space. In order to keep the computational time at an acceptably low level, the number of rays is not increased "orthogonally", that is, instead of adding more rays in each dimension, the existing rays are distributed with respect to this parameter space.

Chapter 10

RADIOSITY METHOD

The radiosity method is based on the numerical solution of the shading equation by the **finite element method**. It subdivides the surfaces into small elemental surface patches. Supposing these patches are small, their intensity distribution over the surface can be approximated by a constant value which depends on the surface and the direction of the emission. We can get rid of this directional dependency if only diffuse surfaces are allowed, since diffuse surfaces generate the same intensity in all directions. This is exactly the initial assumption of the simplest radiosity model, so we are also going to consider this limited case first. Let the energy leaving a unit area of surface i in a unit time in all directions be B_i, and assume that the light density is homogeneous over the surface. This light density plays a crucial role in this model and is also called the **radiosity** of surface i.

The dependence of the intensity on B_i can be expressed by the following argument:

1. Consider a differential dA element of surface A. The total energy leaving the surface dA in unit time is $B \cdot dA$, while the flux in the solid angle $d\omega$ is $d\Phi = I \cdot dA \cdot \cos\phi \cdot d\omega$ if ϕ is the angle between the surface normal and the direction concerned.

2. Expressing the total energy as the integration of the energy contributions over the surface in all directions and assuming diffuse reflection

only, we get:

$$B = \frac{1}{dA} \cdot \int\limits^{2\pi} \frac{d\Phi}{d\omega} \, d\omega = \int\limits^{2\pi} I \cdot \cos\phi \, d\omega = I \cdot \int\limits_{\theta=0}^{2\pi} \int\limits_{\phi=0}^{\pi/2} \cos\phi \sin\phi \, d\theta d\phi = I \cdot \pi$$

$$(10.1)$$

since $d\omega = \sin\phi \, d\phi d\theta$.

Consider the energy transfer of a single surface on a given wavelength. The total energy leaving the surface $(B_i \cdot dA_i)$ can be divided into its own emission and the diffuse reflection of the radiance coming from other surfaces (figure 10.1).
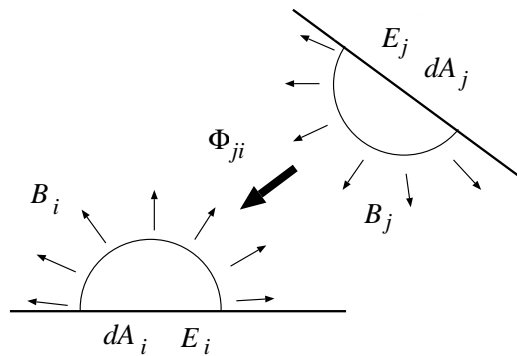
Figure 10.1: Calculation of the radiosity

The emission term is $E_i \cdot dA_i$ if E_i is the emission density which is also assumed to be constant on the surface.

The diffuse reflection is the multiplication of the diffuse coefficient ϱ_i and that part of the energy of other surfaces which actually reaches surface i. Let F_{ji} be a factor, called the **form factor**, which determines that fraction of the total energy leaving surface j which actually reaches surface i.

Considering all the surfaces, their contributions should be integrated, which leads to the following formula of the radiosity of surface i:

$$B_i \cdot dA_i = E_i \cdot dA_i + \varrho_i \cdot \int B_j \cdot F_{ji} \cdot dA_j. \qquad (10.2)$$

Before analyzing this formula any further, some time will be devoted to the meaning and the properties of the form factors.

The fundamental law of photometry (equation 3.15) expresses the energy transfer between two differential surfaces if they are visible from one another. Replacing the intensity by the radiosity using equation 10.1, we get:

$$d\Phi = I \cdot \frac{dA_i \cdot \cos \phi_i \cdot dA_j \cdot \cos \phi_j}{r^2} = B_j \cdot \frac{dA_i \cdot \cos \phi_i \cdot dA_j \cdot \cos \phi_j}{\pi \cdot r^2}. \quad (10.3)$$

If dA_i is not visible from dA_j, that is, another surface is obscuring it from dA_j or it is visible only from the "inner side" of the surface, the energy flux is obviously zero. These two cases can be handled similarly if an indicator variable H_{ij} is introduced:

$$H_{ij} = \begin{cases} 1 \text{ if } dA_i \text{ is visible from } dA_j \\ \\ 0 \text{ otherwise} \end{cases} \quad (10.4)$$

Since our goal is to calculate the energy transferred from one finite surface (ΔA_j) to another (ΔA_i) in unit time, both surfaces are divided into infinitesimal elements and their energy transfer is summed or integrated, thus:

$$\Delta \Phi_{ji} = \int_{\Delta A_i} \int_{\Delta A_j} B_j \cdot H_{ij} \cdot \frac{dA_i \cdot \cos \phi_i \cdot dA_j \cdot \cos \phi_j}{\pi \cdot r^2}. \quad (10.5)$$

By definition, the form factor F_{ji} is a fraction of this energy and the total energy leaving surface j ($B_j \cdot \Delta A_j$):

$$F_{ji} = \frac{1}{\Delta A_j} \cdot \int_{\Delta A_i} \int_{\Delta A_j} H_{ij} \cdot \frac{dA_i \cdot \cos \phi_i \cdot dA_j \cdot \cos \phi_j}{\pi \cdot r^2}. \quad (10.6)$$

It is important to note that the expression of $F_{ji} \cdot \Delta A_j$ is symmetrical with the exchange of i and j, which is known as the **reciprocity relationship**:

$$F_{ji} \cdot \Delta A_j = F_{ij} \cdot \Delta A_i. \quad (10.7)$$

We can now return to the basic radiosity equation. Taking advantage of the homogeneous property of the surface patches, the integral can be replaced by a finite sum:

$$B_i \cdot \Delta A_i = E_i \cdot \Delta A_i + \varrho_i \cdot \sum_j B_j \cdot F_{ji} \cdot \Delta A_j. \quad (10.8)$$

Applying the reciprocity relationship, the term $F_{ji} \cdot \Delta A_j$ can be replaced by $F_{ij} \cdot \Delta A_i$:

$$B_i \cdot \Delta A_i = E_i \cdot \Delta A_i + \varrho_i \cdot \sum_j B_j \cdot F_{ij} \cdot \Delta A_i. \qquad (10.9)$$

Dividing by the area of surface i, we get:

$$B_i = E_i + \varrho_i \cdot \sum_j B_j \cdot F_{ij}. \qquad (10.10)$$

This equation can be written for all surfaces, yielding a linear equation where the unknown components are the surface radiosities (B_i):

$$
\begin{bmatrix}
1 - \varrho_1 F_{11} & -\varrho_1 F_{12} & \dots & -\varrho_1 F_{1N} \\
-\varrho_2 F_{21} & 1 - \varrho_2 F_{22} & \dots & -\varrho_2 F_{2N} \\
\cdot & & & \\
\cdot & & & \\
\cdot & & & \\
-\varrho_N F_{N1} & -\varrho_N F_{N2} & \dots & 1 - \varrho_N F_{NN}
\end{bmatrix}
\begin{bmatrix}
B_1 \\ B_2 \\ \cdot \\ \cdot \\ \cdot \\ B_N
\end{bmatrix}
=
\begin{bmatrix}
E_1 \\ E_2 \\ \cdot \\ \cdot \\ \cdot \\ E_N
\end{bmatrix}
\qquad (10.11)
$$

or in matrix form, having introduced matrix $\mathbf{R}_{ij} = \varrho_i \cdot F_{ij}$:

$$(\mathbf{1} - \mathbf{R}) \cdot \mathbf{B} = \mathbf{E} \qquad (10.12)$$

($\mathbf{1}$ stands for the unit matrix).

The meaning of F_{ii} is the fraction of the energy reaching the very same surface. Since in practical applications the elemental surface patches are planar polygons, F_{ii} is 0.

Both the number of unknown variables and the number of equations are equal to the number of surfaces (N). The solution of this linear equation is, at least theoretically, straightforward (we shall consider its numerical aspects and difficulties later). The calculated B_i radiosities represent the light density of the surface on a given wavelength. Recalling Grassman's laws, to generate color pictures at least three independent wavelengths should be selected (say red, green and blue), and the color information will come from the results of the three different calculations.

Thus, to sum up, the basic steps of the **radiosity method** are these:

1. F_{ij} form factor calculation.

2. Describe the light emission (E_i) on the representative wavelengths, or in the simplified case on the wavelength of red, green and blue colors. Solve the linear equation for each representative wavelength, yielding $B_i^{\lambda_1}$, $B_i^{\lambda_2}$ ... $B_i^{\lambda_n}$.

3. Generate the picture taking into account the camera parameters by any known hidden surface algorithm. If it turns out that surface i is visible in a pixel, the color of the pixel will be proportional to the calculated radiosity, since the intensity of a diffuse surface is proportional to its radiosity (equation 10.1) and is independent of the direction of the camera.

Constant color of surfaces results in the annoying effect of faceted objects, since the eye psychologically accentuates the discontinuities of the color distribution. To create the appearance of smooth surfaces, the tricks of Gouraud shading can be applied to replace the jumps of color by linear changes. In contrast to Gouraud shading as used in incremental methods, in this case vertex colors are not available to form a set of knot points for interpolation. These vertex colors, however, can be approximated by averaging the colors of adjacent polygons (see figure 10.2).

$$B_v = \frac{B_1 + B_2 + B_3 + B_4}{4}$$
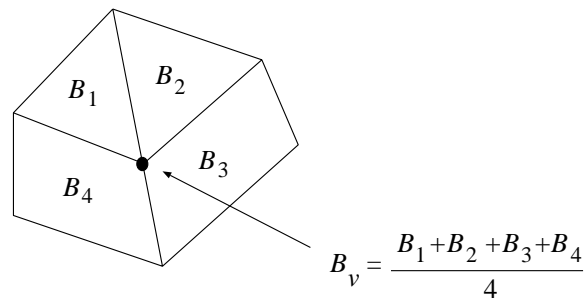
Figure 10.2: Color interpolation for images created by the radiosity method

Note that the first two steps of the radiosity method are independent of the actual view, and the form factor calculation depends only on the

geometry of the surface elements. In camera animation, or when the scene is viewed from different perspectives, only the third step has to be repeated; the computationally expensive form factor calculation and the solution of the linear equation should be carried out only once for a whole sequence. In addition to this, the same form factor matrix can be used for sequences, when the lightsources have time varying characteristics.

10.1   Form factor calculation

The most critical issue in the radiosity method is efficient form factor calculation, and thus it is not surprising that considerable research effort has gone into various algorithms to evaluate or approximate the formula which defines the form factors:

$$F_{ij} = \frac{1}{\Delta A_i} \cdot \int\limits_{\Delta A_i} \int\limits_{\Delta A_j} H_{ij} \cdot \frac{dA_i \cdot \cos \phi_i \cdot dA_j \cdot \cos \phi_j}{\pi \cdot r^2}. \qquad (10.13)$$

As in the solution of the shading problem, the different solutions represent different compromises between the conflicting objectives of high calculation speed, accuracy and algorithmic simplicity.

In our survey the various approaches are considered in order of increasing algorithmic complexity, which, interestingly, does not follow the chronological sequence of their publication.

10.1.1   Randomized form factor calculation

The randomized approach is based on the recognition that the formula defining the form factors can be taken to represent the probability of a quite simple event if the underlying probability distributions are defined properly.

An appropriate such event would be a surface j being hit by a particle leaving surface i. Let us denote the event that a particle leaves surface dA_i by $PLS(dA_i)$. Expressing the probability of the "hit" of surface j by the total probability theorem we get:

$$\Pr\{\text{hit } \Delta A_j\} = \int\limits_{\Delta A_i} \Pr\{\text{hit } \Delta A_j \mid PLS(dA_i)\} \cdot \Pr\{PLS(dA_i)\}. \quad (10.14)$$

The hitting of surface j can be broken down into the separate events of hitting the various differential elements dA_j composing ΔA_j. Since hitting of dA_k and hitting of dA_l are exclusive events if $dA_k \neq dA_l$:

$$\Pr\{\text{hit } \Delta A_j \mid PLS(dA_i)\} = \int_{\Delta A_j} \Pr\{\text{hit } dA_j \mid PLS(dA_i)\}. \qquad (10.15)$$

Now the probability distributions involved in the equations are defined:

1. Assume the origin of the particle to be selected randomly by uniform distribution:
$$\Pr\{PLS(dA_i)\} = \frac{1}{\Delta A_i} \cdot dA_i. \qquad (10.16)$$

2. Let the direction in which the particle leaves the surface be selected by a distribution proportional to the cosine of the angle between the direction and the surface normal:

$$\Pr\{\text{particle leaves in solid angle } d\omega\} = \frac{\cos \phi_i \cdot d\omega}{\pi}. \qquad (10.17)$$

The denominator π guarantees that the integration of the probability over the whole hemisphere yields 1, hence it deserves the name of probability density function. Since the solid angle of dA_j from dA_i is $dA_j \cdot \cos \phi_j / r^2$ where r is the distance between dA_i and dA_j, and ϕ_j is the angle of the surface normal of dA_j and the direction of dA_i, the probability of equation 10.15 is:
$$\Pr\{\text{hit } dA_j \mid PLS(dA_i)\} =$$

$$\Pr\{dA_j \text{ is not hidden from } dA_i \wedge \text{ particle leaves in the solid angle of } dA_j\}$$

$$= \frac{H_{ij} \cdot dA_j \cdot \cos \phi_j \cdot \cos \phi_i}{r^2 \cdot \pi} \qquad (10.18)$$

where H_{ij} is the indicator function of the event "dA_j is visible from dA_i".
Substituting these into the original probability formula:

$$\Pr\{\text{hit}\} = \frac{1}{\Delta A_i} \cdot \int_{\Delta A_i} \int_{\Delta A_j} H_{ij} \cdot \frac{dA_i \cdot \cos \phi_i \cdot dA_j \cdot \cos \phi_j}{\pi \cdot r^2}. \qquad (10.19)$$

This is exactly the same as the formula for form factor F_{ij}. This probability, however, can be estimated by random simulation. Let us generate n

particles randomly using uniform distribution on the surface i to select the origin, and a cosine density function to determine the direction. The origin and the direction define a ray which may intersect other surfaces. That surface will be hit whose intersection point is the closest to the surface from which the particle comes. If shooting n rays randomly surface j has been hit k_j times, then the probability or the form factor can be estimated by the relative frequency:

$$F_{ij} \approx \frac{k_j}{n}. \tag{10.20}$$

Two problems have been left unsolved:

- How can we select n to minimize the calculations but to sustain a given level of accuracy?

- How can we generate uniform distribution on a surface and cosine density function in the direction?

Addressing the problem of the determination of the necessary number of attempts, we can use the laws of large numbers.

The inequality of Bernstein and Chebyshev [Rén81] states that if the absolute value of the difference of the event frequency and the probability is expected not to exceed ϵ with probability δ, then the minimum number of attempts (n) is:

$$n \geq \frac{9 \log 2/\delta}{8\epsilon^2}. \tag{10.21}$$

The generation of random distributions can rely on random numbers of uniform distribution in $[0..1]$ produced by the pseudo-random algorithm of programming language libraries. Let the probability distribution function of the desired distribution be $P(x)$. A random variable x which has $P(x)$ probability distribution can be generated by transforming the random variable r that is uniformly distributed in $[0..1]$ applying the following transformation:

$$x = P^{-1}(r). \tag{10.22}$$

10.1.2 Analytic and geometric methods

The following algorithms focus first on the inner section of the double integration, then estimate the outer integration. The inner integration is given some geometric interpretation which is going to be the base of the calculation. This inner integration has the following form:

$$d_i F_{ij} = \int\limits_{\Delta A_j} H_{ij} \cdot \frac{\cos \phi_i \cdot \cos \phi_j}{\pi \cdot r^2} \, dA_j. \qquad (10.23)$$
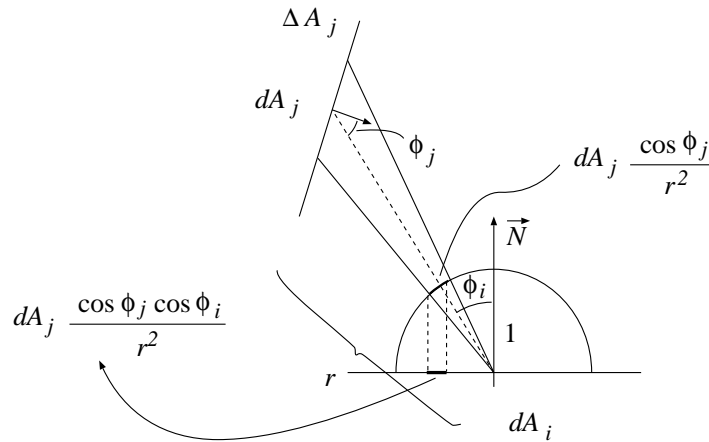
Figure 10.3: Geometric interpretation of hemisphere form factor algorithm

Nusselt [SH81] has realized that this formula can be interpreted as projecting the visible parts of ΔA_j onto the unit hemisphere centered above dA_i, then projecting the result orthographically onto the base circle of this hemisphere in the plane of dA_i (see figure 10.3), and finally calculating the ratio of the doubly projected area and the area of the unit circle (π). Due to the central role of the unit hemisphere, this method is called the **hemisphere algorithm**.

Later Cohen and Greenberg [CG85] have shown that the projection calculation can be simplified, and more importantly, supported by image synthesis hardware, if the hemisphere is replaced by a half cube. Their method is called the **hemicube algorithm**.

Beran-Koehn and Pavicic have demonstrated in their recent publication [BKP91] that the necessary calculations can be significantly decreased if a **cubic tetrahedron** is used instead of the hemicube.

Having calculated the inner section of the integral, the outer part must be evaluated. The simplest way is to suppose that it is nearly constant on ΔA_i, so the outer integral is estimated as the multiplication of the inner integral at the middle of ΔA_i and the area of this surface element:

$$F_{ij} = \frac{1}{\Delta A_i} \int\limits_{\Delta A_i} d_i F_{ij} \, dA_i \approx d_i F_{ij} = \int\limits_{\Delta A_j} H_{ij} \cdot \frac{\cos \phi_i \cdot \cos \phi_j}{\pi \cdot r^2} \, dA_j. \quad (10.24)$$

More accurate computations require the evaluation of the inner integral in several points on ΔA_i and some sort of numerical integration technique should be used for the integral calculation.

10.1.3   Analytic form factor computation

The inner section of the form factor integral, or as it is called the form factor between a finite and differential area, can be written as a surface integral in a vector space, denoting the vector between dA_i and dA_j by $\vec{r}$, the unit normal to dA_i by $\vec{n}_i$, and the surface element vector $\vec{n}_j \cdot dA_j$ by $d\vec{A}_j$:

$$d_i F_{ij} = \int\limits_{\Delta A_j} H_{ij} \cdot \frac{\cos \phi_i \cdot \cos \phi_j}{\pi \cdot r^2} \, dA_j = - \int\limits_{\Delta A_j} H_{ij} \cdot \frac{(\vec{n}_i \cdot \vec{r})}{\pi \cdot |\vec{r}|^4} \cdot \vec{r} \, d\vec{A}_j = \int\limits_{\Delta A_j} \vec{w} \, d\vec{A}_j.$$
$$(10.25)$$

If we could find a vector field $\vec{v}$, such that $rot \ \vec{v} = \vec{w}$, the area integral could be transformed into the contour integral $\int \vec{v} d\vec{l}$ by Stoke's theorem. This idea has been followed by Hottel and Sarofin [HS67], and they were successful in providing a formula for the case when there are no occlusions, or the visibility term H_{ij} is everywhere 1:

$$d_i F_{ij} = \frac{1}{2\pi} \sum_{l=0}^{L-1} \frac{\text{angle}(\vec{R}_l, \vec{R}_{l\oplus 1})}{|\vec{R}_l \times \vec{R}_{l\oplus 1}|} (\vec{R}_l \times \vec{R}_{l\oplus 1}) \cdot \vec{n}_i \qquad (10.26)$$

where

1. $\text{angle}(\vec{a}, \vec{b})$ is the signed angle between two vectors. The sign is positive if $\vec{b}$ is rotated clockwise from $\vec{a}$ looking at them in the opposite direction to $\vec{n}_i$,

2. $\oplus$ represents addition modulo L. It is a circular next operator for vertices,

3. L is the number of vertices of surface element j,

4. $\vec{R}_l$ is the vector from the differential surface i to the lth vertex of the surface element j.

We do not aim to go into the details of the original derivation of this formula based on the theory of vector fields, because it can also be proven relying on geometric considerations of the hemispherical projection.

10.1.4   Hemisphere algorithm

First of all the result of Nusselt is proven using figure 10.3, which shows that the inner form factor integral can be calculated by a double projection of ΔA_j, first onto the unit hemisphere centered above dA_i, then to the base circle of this hemisphere in the plane of dA_i, and finally by calculating the ratio of the double projected area and the area of the unit circle (π). By geometric arguments, or by the definition of solid angles, the projected area of a differential area dA_j on the surface of the hemisphere is $dA_j \cdot \cos \phi_j / r^2$. This area is orthographically projected onto the plane of dA_i, multiplying the area by factor $\cos \phi_i$. The ratio of the double projected area and the area of the base circle is:

$$\frac{\cos \phi_i \cdot \cos \phi_j}{\pi \cdot r^2} \cdot dA_j. \tag{10.27}$$

Since the double projection is a one-to-one mapping, if surface ΔA_j is above the plane of A_i, the portion, taking the whole ΔA_j surface into account, is:

$$\int\limits_{\Delta A_j} H_{ij} \cdot \frac{\cos \phi_i \cdot \cos \phi_j}{\pi \cdot r^2} \, dA_j = d_i F_{ij}. \tag{10.28}$$

This is exactly the formula of an inner form factor integral.

Now we turn to the problem of the **hemispherical projection** of a planar polygon. To simplify the problem, consider only one edge line of the polygon first, and two vertices, $\vec{R}_l$ and $\vec{R}_{l\oplus 1}$, on it (figure 10.4). The hemispherical projection of this line is a half great circle. Since the radius
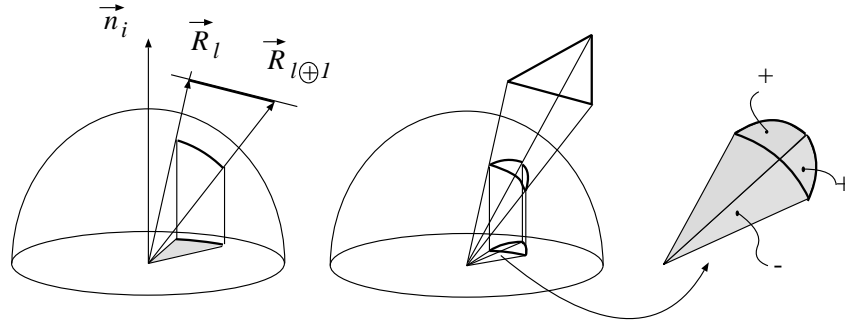
Figure 10.4: Hemispherical projection of a planar polygon

of this great circle is 1, the area of the sector formed by the projections of $\vec{R}_l$ and $\vec{R}_{l\oplus1}$ and the center of the hemisphere is simply half the angle of $\vec{R}_l$ and $\vec{R}_{l\oplus1}$. Projecting this sector orthographically onto the equatorial plane, an ellipse sector is generated, having the area of the great circle sector multiplied by the cosine of the angle of the surface normal $\vec{n}_i$ and the normal of the segment $(\vec{R}_l \times \vec{R}_{l\oplus1})$.

The area of the doubly projected polygon can be obtained by adding and subtracting the areas of the ellipse sectors of the different edges, as is demonstrated in figure 10.4, depending on whether the projections of vectors $\vec{R}_l$ and $\vec{R}_{l\oplus1}$ follow each other clockwise. This sign value can also be represented by a signed angle of the two vectors, expressing the area of the double projected polygon as a summation:

$$\sum_{l=0}^{L-1} \frac{1}{2} \cdot \text{angle}(\vec{R}_l, \vec{R}_{l\oplus1}) \frac{(\vec{R}_l \times \vec{R}_{l\oplus1})}{|\vec{R}_l \times \vec{R}_{l\oplus1}|} \cdot \vec{n}_i. \tag{10.29}$$

Having divided this by π to calculate the ratio of the area of the double projected polygon and the area of the equatorial circle, equation 10.26 can be generated.

These methods have supposed that surface ΔA_j is above the plane of dA_i and is totally visible. Surfaces below the equatorial plane do not pose any problems, since we can get rid of them by the application of a clipping algorithm. Total visibility, that is when visibility term H_{ij} is everywhere 1,

however, is only an extreme case in the possible arrangements. The other extreme case is when the visibility term is everywhere 0, and thus the form factor will obviously be zero.

When partial occlusion occurs, the computation can make use of these two extreme cases according to the following approaches:

1. A continuous (object precision) visibility algorithm is used in the form factor computation to select the visible parts of the surfaces. Having executed this step, the parts are either totally visible or hidden from the given point on surface i.

2. The visibility term is estimated by firing several rays to surface element j and averaging their 0/1 associated visibilities. If the result is about 1, no occlusion is assumed; if it is about 0, the surface is assumed to be obscured; otherwise the surface i has to be subdivided, and the whole step repeated recursively [Tam92].

10.1.5 Hemicube algorithm

The hemicube algorithm is based on the fact that it is easier to project onto a planar rectangle than onto a spherical surface. Due to the change of the underlying geometry, the double projection cannot be expected to provide the same results as for a hemisphere, so in order to evaluate the inner form factor integral some corrections must be made during the calculation. These correction parameters are generated by comparing the needed terms and the terms resulting from the hemicube projections.

Let us examine the projection onto the top of the hemicube. Using geometric arguments and the notations of figure 10.5, the projected area of a differential patch dA_j is:

$$T(dA_j) = H_{ij} \cdot \left(\frac{R}{r}\right)^2 \cdot dA_j \cdot \frac{\cos \phi_j}{\cos \phi_i} = H_{ij} \cdot \frac{dA_j \cdot \cos \phi_j \cdot \cos \phi_i}{\pi \cdot r^2} \cdot \frac{\pi}{(\cos \phi_i)^4} \tag{10.30}$$

since $R = 1/\cos \phi_i$.

Looking at the form factor formula, we notice that a weighted area is to be calculated, where the weight function compensates for the unexpected $\pi/(\cos \phi_i)^4$ term. Introducing the compensating function w_z valid on the top of the hemicube, and expressing it by geometric considerations of figure 10.5
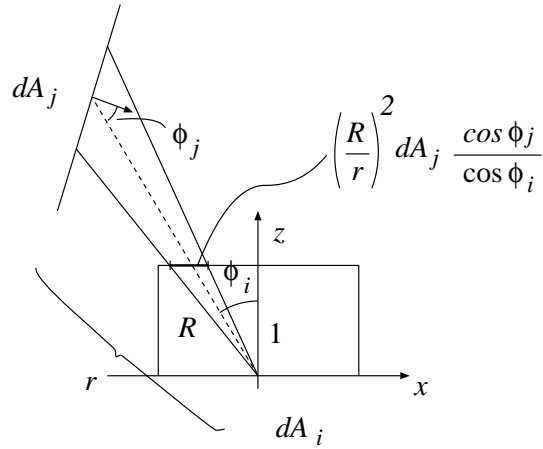
Figure 10.5: Form factor calculation by hemicube algorithm

which supposes an (x, y, z) coordinate system attached to the dA_i, with axes parallel with the sides of the hemicube, we get:

$$w_z(x, y) = \frac{(\cos \phi_i)^4}{\pi} = \frac{1}{\pi (x^2 + y^2 + 1)^2}. \tag{10.31}$$

Similar considerations can lead to the calculation of the correction terms of the projection on the side faces of the hemicube:

If the side face is perpendicular to the y axis, then:

$$w_y(x, z) = \frac{z}{\pi (x^2 + z^2 + 1)^2} \tag{10.32}$$

or if the side face is perpendicular to the x axis:

$$w_x(y, z) = \frac{z}{\pi (z^2 + y^2 + 1)^2}. \tag{10.33}$$

The weighted area defining the inner form factor is an area integral of a weight function. If ΔA_j has a projection onto the top of the hemicube only, then:

$$d_i F_{ij} = \int\limits_{\Delta A_j} T(dA_j) \cdot \frac{\cos^4 \phi_i}{\pi}. \tag{10.34}$$

Instead of integrating over ΔA_j, the same integral can also be calculated on the top of the hemicube in an x, y, z coordinate system:

$$d_i F_{ij}^{\text{top}} = \int\limits_{T(\Delta A_j)} H_{ij}(x, y) \cdot \frac{1}{\pi(x^2 + y^2 + 1)^2} \, dxdy \qquad (10.35)$$

since $\cos \phi_i = 1/(x^2 + y^2 + 1)^{1/2}$. Indicator $H_{ij}(x, y)$ shows whether ΔA_j is really visible through hemicube point $(x, y, 1)$ from ΔA_i or if it is obscured.

This integral is approximated by a finite sum having generated a $P \times P$ raster mesh on the top of the hemicube.

$$d_i F_{ij}^{\text{top}} = \int\limits_{T(\Delta A_j)} H_{ij}(x, y) \cdot w_z(x, y) dxdy \approx$$

$$\sum_{X=-P/2}^{P/2-1} \sum_{Y=-P/2}^{P/2-1} H_{ij}(X, Y) \cdot w_z(X, Y) \frac{1}{P^2}. \qquad (10.36)$$

The only unknown term here is H_{ij}, which tells us whether or not surface j is visible through the raster cell called "pixel" (X, Y). Thanks to the research that has been carried out into hidden surface problems there are many effective algorithms available which can also be used here. An obvious solution is the application of simple ray tracing. The center of dA_i and the pixel defines a ray which may intersect several other surfaces. If the closest intersection is on the surface j, then $H_{ij}(X, Y)$ is 1, otherwise it is 0.

A faster solution is provided by the z-buffer method. Assume the color of the surface ΔA_j to be j, the center of the camera to be dA_i and the 3D window to be a given face of the hemicube. Having run the z-buffer algorithm, the pixels are set to the "color" of the surfaces visible in them. Taking advantage of the above definition of color (color is the index of the surface), each pixel will provide information as to which surface is visible in it. We just have to add up the weights of those pixels which contain "color" j in order to calculate the differential form factor $d_i F_{ij}^{\text{top}}$.

The projections on the side faces can be handled in exactly the same way, except that the weight function has to be selected differently (w_x or w_y depending on the actual side face). The form factors are calculated as a sum of contributions of the top and side faces.

The complete algorithm, to calculate the F_{ij} form factors using the z-buffer method, is:

```
for i = 1 to N do for j = 1 to N do F_ij = 0;
for i = 1 to N do
    camera = center of ΔA_i;
    for k = 1 to 5 do              // consider each face of the hemicube
        window = kth face of the hemicube;
        for x = 0 to P − 1 do
            for y = 0 to P − 1 do pixel[x, y] = 0;
        Z-BUFFER ALGORITHM (color of surface j is j)
        for x = 0 to P − 1 do for y = 0 to P − 1 do
            if (pixel[x, y] > 0) then
                F_{i,pixel[x,y]} += w_k(x − P/2, y − P/2)/P²;
    endfor
    endfor
endfor
```

In the above algorithm the weight function $w_k(x - P/2, y - P/2)/P^2$ must be evaluated for those pixels through which other surfaces are visible and must be added to that form factor which corresponds to the visible surface. This is why values of weight functions at pixel centers are called **delta form factors**. Since the formula for weight functions contains many multiplications and a division, its calculation in the inner loop of the algorithm can slow down the form factor computation. However, these weight functions are common to all hemicubes, thus they must be calculated only once and then stored in a table which can be re-used whenever a value of the weight function is needed.

Since the z-buffer algorithm has $O(N \cdot P^2)$ worst case complexity, the computation of the form factors, embedding $5N$ z-buffer steps, is obviously $O(N^2 \cdot P^2)$, where N is the number of surface elements and P^2 is the number of pixels in the z-buffer. It is important to note that P can be much less than the resolution of the screen, since now the "pixels" are used only to approximate an integral finitely. Typical values of P are $50 \ldots 200$.

Since the z-buffer step can be supported by a hardware algorithm this approach is quite effective on workstations supported by graphics accelerators.

10.1.6   Cubic tetrahedral algorithm

The hemicube algorithm replaced the hemisphere by a half cube, allowing the projection to be carried out on five planar rectangles, or side faces of the cube, instead of on a spherical surface. The number of planar surfaces can be decreased by using a cubic tetrahedron as an intermediate surface [BKP91], [BKP92].
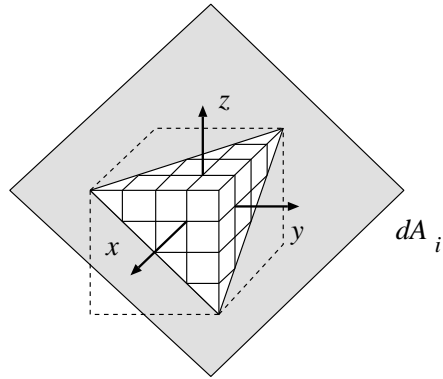
Figure 10.6: Cubic tetrahedral method

An appropriate cubic tetrahedron may be constructed by slicing a cube by a plane that passes through three of its vertices, and placing the generated pyramid on surface i (see figure 10.6). A convenient coordinate system is defined with axes perpendicular to the faces of the tetrahedron, and setting scales to place the apex in point $[1, 1, 1]$. The base of the tetrahedron will be a triangle having vertices at $[1, 1, -2]$, $[1, -2, 1]$ and $[-2, 1, 1]$.

Consider the projection of a differential surface dA_j on a side face perpendicular to x axis, using the notations of figure 10.7. The projected area is:

$$dA'_j = \frac{dA_j \cdot \cos \phi_j}{\cos\theta} \cdot \frac{|\vec{R}|^2}{r^2}.\qquad(10.37)$$

The correction term, to provide the internal variable in the form factor integral, is:

$$\frac{dA_j \cdot \cos \phi_j \cdot \cos \phi_i}{\pi \cdot r^2} = dA'_j \cdot \frac{\cos \phi_i \cdot \cos \theta}{\pi \cdot |\vec{R}|^2} = dA'_j \cdot w(\vec{R}).\qquad(10.38)$$
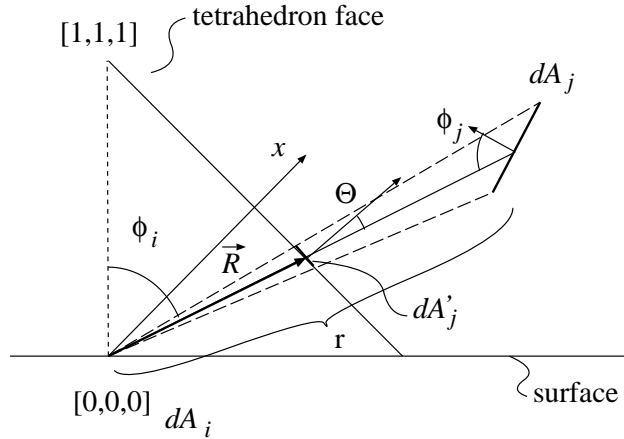
Figure 10.7: Projection to the cubic tetrahedron

Expressing the cosine of angles by a scalar product with $\vec{R}$ pointing to the projected area:

$$\cos\theta = \frac{\vec{R}\cdot[1,0,0]}{|\vec{R}|}, \qquad \cos\phi_i = \frac{\vec{R}\cdot[1,1,1]}{|\vec{R}|\cdot|[1,1,1]|}. \tag{10.39}$$

Vector $\vec{R}$ can also be defined as the sum of the vector pointing to the apex of the pyramid ($[1,1,1]$) and a linear combination of side vectors of pyramid face perpendicular to x axis:

$$\vec{R} = [1,1,1] + (1-u)\cdot[0,-1,0] + (1-v)\cdot[0,0,-1] = [1,u,v]. \tag{10.40}$$

This can be turned to the previous equation first, then to the formula of the correction term:

$$w(u,v) = \frac{u+v+1}{\pi\cdot\sqrt{3}\cdot(u^2+v^2+1)^2}. \tag{10.41}$$

Because of symmetry, the values of this weight function — that is the delta form factors — need to be computed and stored for only one-half of any face when the delta form factor table is generated. It should be mentioned that cells located along the base of the tetrahedron need special

treatment, since they have triangular shape. They can either be simply ignored, because their delta form factors are usually very small, or they can be evaluated for the center of the triangle instead of the center of the rectangular pixel.

10.2 Solution of the linear equation

The most obvious way to solve a linear equation is to apply the Gauss elimination method [PFTV88]. Unfortunately it fails to solve the radiosity equation for more complex models effectively, since it has $O(N^3)$ complexity, and also it accumulates the round of errors of digital computers and magnifies these errors to the extent that the matrix is close to singular.

Fortunately another technique, called **iteration**, can overcome both problems. Examining the radiosity equation,

$$B_i = E_i + \varrho_i \sum_j B_j \cdot F_{ij}$$

we will see that it gives the equality of the energy which has to be radiated due to emission and reflection (right side) and the energy really emitted (left side). Suppose that only estimates are available for B_j radiosities, not exact values. These estimates can be regarded as right side values, thus having substituted them into the radiosity equation, better estimates can be expected on the left sides. If these estimates were exact — that is they satisfied the radiosity equation —, then the iteration would not alter the radiosity values. Thus, if this iteration converges, its limit will be the solution of the original radiosity equation.

In order to examine the method formally, the matrix version of the radiosity equation is used to describe a single step of the iteration:

$$\mathbf{B}(m+1) = \mathbf{R} \cdot \mathbf{B}(m) + \mathbf{E}. \tag{10.42}$$

A similar equation holds for the previous iteration too. Subtracting the two equations, and applying the same consideration recursively, we get:

$$\mathbf{B}(m+1) - \mathbf{B}(m) = \mathbf{R} \cdot (\mathbf{B}(m) - \mathbf{B}(m-1)) = \mathbf{R^m} \cdot (\mathbf{B}(1) - \mathbf{B}(0)). \tag{10.43}$$

The iteration converges if

$$\lim_{m \to \infty} \|\mathbf{B}(m+1) - \mathbf{B}(m)\| = 0, \quad \text{that is if} \quad \lim_{m \to \infty} \|\mathbf{R^m}\| = 0$$

for some matrix norm. Let us use the $\|\mathbf{R}\|_\infty$ norm defined as the maximum of absolute row sums

$$\|\mathbf{R}\|_\infty = \max_i \{\sum_j F_{ij} \cdot \varrho_i\} \tag{10.44}$$

and a vector norm that is compatible with it:

$$\|\mathbf{b}\|_\infty = \max_i \{|b_i|\}. \tag{10.45}$$

Denoting $\|\mathbf{R}\|$ by q, we have:

$$\|\mathbf{B}(m+1) - \mathbf{B}(m)\| = \|\mathbf{R^m} \cdot (\mathbf{B}(1) - \mathbf{B}(0))\| \leq \|\mathbf{R}\|^m \cdot \|\mathbf{B}(1) - \mathbf{B}(0)\| =$$

$$q^m \cdot \|\mathbf{B}(1) - \mathbf{B}(0)\| \tag{10.46}$$

according to the properties of matrix norms. Since F_{ij} represents the portion of the radiated energy of surface i, which actually reaches surface j, $\sum_j F_{ij}$ is that portion which is radiated towards any other surface. This obviously cannot exceed 1, and for physically correct models, diffuse reflectance $\varrho_i < 1$, giving a norm that is definitely less than 1. Consequently $q < 1$, which provides the convergence with, at least, the speed of a geometric series.

The complexity of the iteration solution depends on the operations needed for a single step and the number of iterations providing convergence. A single step of the iteration requires the multiplication of an N dimensional vector and an $N \times N$ dimensional matrix, which requires $O(N^2)$ operations.

Concerning the number of necessary steps, we concluded that the speed of the convergence is at least geometric by a factor $q = \|\mathbf{R}\|_\infty$. The infinite norm of $\mathbf{R}$ is close to being independent of the number of surface elements, since as the number of surface elements increases, the value of form factors decreases, sustaining a constant sum of rows, representing that portion of the energy radiated by surface i, which is gathered by other surfaces, multiplied by the diffuse coefficient of surface i. Consequently, the number of necessary iterations is independent of the number of surface elements, making the iteration solution an $O(N^2)$ process.

10.2.1   Gauss–Seidel iteration

The convergence of the iteration can be improved by the method of Gauss–Seidel iteration. Its basic idea is to use the new iterated values immediately

when they are available, and not to postpone their usage until the next iteration step. Consider the calculation of B_i in the normal iteration:

$$B_i(m + 1) = E_i + R_{i,1} \cdot B_1(m) + R_{i,2} \cdot B_2(m) + ... + R_{i,N} \cdot B_N(m). \quad (10.47)$$

During the calculation of $B_i(m + 1)$, values $B_1(m + 1), ..., B_{i-1}(m + 1)$ have already been calculated, so they can be used instead of their previous value, modifying the iteration, thus:

$$B_i(m + 1) = E_i + R_{i,1} \cdot B_1(m + 1) + ... + R_{i,i-1} \cdot B_{i-1}(m + 1) +$$

$$R_{i,i+1} \cdot B_{i+1}(m) + ... + R_{i,N} \cdot B_N(m) \quad (10.48)$$

(recall that $R_{i,i} = 0$ in the radiosity equation).

A trick, called **successive relaxation**, can further improve the speed of convergence. Suppose that during the mth step of the iteration the radiosity vector $\mathbf{B}(m + 1)$ was computed. The difference from the previous estimate is:

$$\Delta \mathbf{B} = \mathbf{B}(m + 1) - \mathbf{B}(m) \quad (10.49)$$

showing the magnitude of difference, as well as the direction of the improvement in N dimensional space. According to practical experience, the direction is quite accurate, but the magnitude is underestimated, requiring the correction by a relaxation factor ω:

$$\mathbf{B}^*(m + 1) = \mathbf{B}(m) + \omega \cdot \Delta \mathbf{B}. \quad (10.50)$$

The determination of ω is a crucial problem. If it is too small, the convergence will be slow; if it is too great, the system will be unstable and divergent. For many special matrices, the optimal relaxation factors have already been determined, but concerning our radiosity matrix, only practical experiences can be relied on. Cohen [CGIB86] suggests that relaxation factor 1.1 is usually satisfactory.

10.3 Progressive refinement

The previously discussed radiosity method determined the form factor matrix first, then solved the linear equation by iteration. Both steps require $O(N^2)$ time and space, restricting the use of this algorithm in commercial

applications. Most of the form factors, however, have very little effect on
the final image, thus, if they were taken to be 0, a great amount of time
and space could be gained for the price of a negligible deterioration of the
image quality. A criterion for selecting unimportant form factors can be
established by the careful analysis of the iteration solution of the radiosity
equation:

$$B_i(m+1) = E_i + \varrho_i \sum_j B_j(m) \cdot F_{ij} = E_i + \sum_j \left(B_i \text{ due to } B_j(m)\right)$$

$$(B_i \text{ due to } B_j) = \varrho_i \cdot B_j \cdot F_{ij}. \qquad (10.51)$$

If B_j is small, then the whole column i of $\mathbf{R}$ will not make too much
difference, thus it is not worth computing and storing its elements. This
seems acceptable, but how can we decide which radiosities will be small, or
which part of matrix $\mathbf{R}$ should be calculated, before starting the iteration?
We certainly cannot make the decision before knowing something about the
radiosities, but we can definitely do it during the iteration by calculating a
column of the form factor matrix only when it turns out that it is needed,
since the corresponding surface has significant radiosity.

Suppose we have an estimate B_j allowing for the calculation of the contri-
bution of this surface to all the others, and for determining a better estimate
for other surfaces by adding this new contribution to their estimated value.
If an estimate B_j increases by ΔB_j, due to the contribution of other surfaces
to this radiosity, other surface radiosities should also be corrected accord-
ing to the new contribution of B_j, resulting in an iterative and progressive
refinement of surface radiosities:

$$B_i^{\text{new}} = B_i^{\text{old}} + \varrho_i \cdot (\Delta B_j) \cdot F_{ij}. \qquad (10.52)$$

Note that, in contrast to the previous radiosity method when we were
interested in how a surface gathers energy from other surfaces, now the di-
rection of the light is followed focusing on how surfaces shoot light to other
surfaces. A radiosity increment of a surface, which has not yet been used
to update other surface radiosities, is called **unshot radiosity**. In fact, in
equation 10.52, the radiosity of other surfaces should be corrected according
to the unshot radiosity of surface j. It seems reasonable to select for shoot-
ing that surface which has the highest unshot radiosity. Having selected a

surface, the corresponding column of the form factor matrix should be calculated. We can do that on every occasion when a surface is selected to shoot its radiosity. This reduces the burden of the storage of the $N \times N$ matrix elements to only a single column containing N elements, but necessitates the recalculation of the form factors. Another alternative is to store the already generated columns, allowing for reduction of the storage requirements by omitting those columns whose surfaces are never selected, due to their low radiosity. Let us realize that equation 10.52 requires $F_{1j}, F_{2j}, \ldots, F_{Nj}$, that is a single column of the form factor matrix, to calculate the radiosity updates due to ΔB_j. The hemicube method, however, supports "parallel" generation of the rows of the form factor matrix, not of the columns. For different rows, different hemicubes have to be built around the surfaces. Fortunately, the reciprocity relationship can be applied to evaluate a single column of the matrix based on a single hemicube:

$$F_{ji} \cdot \Delta A_j = F_{ij} \cdot \Delta A_i \quad \Longrightarrow \quad F_{ij} = F_{ji} \cdot \frac{\Delta A_j}{\Delta A_i} \quad (i = 1, ..., N) \qquad (10.53)$$

These considerations have formulated an iterative algorithm, called **progressive refinement**. The algorithm starts by initializing the total (B_i) and unshot (U_i) radiosities of the surfaces to their emission, and stops if the unshot radiosity is less than an acceptable threshold for all the surfaces:

> **for** $j = 1$ **to** N **do** $B_j = E_j$; $U_j = E_j$
> **do**
> j = Index of the surface of maximum U_j;
>   Calculate F_{j1}, F_{j2} ,..., F_{jN} by a single hemicube;
>   **for** $i = 1$ **to** N **do**
> $\Delta B_i = \varrho_i \cdot U_j \cdot F_{ji} \cdot \Delta A_j / \Delta A_i$;
> $U_i \mathrel{+}= \Delta B_i$;
> $B_i \mathrel{+}= \Delta B_i$;
>   **endfor**
> $U_j = 0$;
> $error = \max\{U_1, U_2, ..., U_N\}$;
> **while** $error > threshold$;

This algorithm is always convergent, since the total amount of unshot energy decreases in each step by an attenuation factor of less than 1. This

statement can be proven by examining the total unshot radiosities during the iteration, supposing that U_j was maximal in step m, and using the notation $q = \|\mathbf{R}\|_\infty$ again:

$$\sum_i^N U_i(m+1) = \sum_{i\neq j}^N U_i(m) + U_j \cdot \sum_i^N \varrho_i \cdot F_{ij} = (\sum_i^N U_i(m)) - U_j + U_j \sum_i^N \varrho_i \cdot F_{ij} \le$$

$$\le (\sum_i^N U_i(m)) - (1-q)\cdot U_j \le (1 - \frac{1-q}{N})\cdot \sum_i^N U_i(m) = q^* \cdot \sum_i^N U_i(m) \quad (10.54)$$

since $q = \max_i\{\sum_i^N \varrho_i \cdot F_{ij}\} < 1$ and $U_j \ge \sum_i^N U_i/N$, because it is the maximal value among U_i-s.

Note that, in contrast to the normal iteration, the attenuation factor q^* defining the speed of convergence now does depend on N, slowing down the convergence by approximately N times, and making the number of necessary iterations proportional to N. A single iteration contains a single loop of length N in progressive refinement, resulting in $O(N^2)$ overall complexity, taking into account the expected number of iterations as well. Interestingly, progressive refinement does not decrease the $O(N^2)$ time complexity, but in its simpler form when the form factor matrix is not stored, it can achieve $O(N)$ space complexity instead of the $O(N^2)$ behavior obtained by the original method.

10.3.1   Application of vertex-surface form factors

In the traditional radiosity and the discussed progressive refinement methods, the radiosity distributions of the elemental surfaces were assumed to be constant, as were the normal vectors. This is obviously far from accurate, and the effects need to be reduced by a bilinear interpolation of Gouraud shading at the last step of the image generation. In progressive refinement, however, the linear radiosity approximation can be introduced earlier, even during the phase of the calculation of radiosities. Besides, the real surface normals in the vertices of the approximating polygons can be used resulting in a more accurate computation.

This method is based on the examination of energy transfer between a differential area (dA_i) around a vertex of a surface and another finite surface (ΔA_j), and concentrates on the radiosity of vertices of polygons instead of

the radiosities of the polygons themselves. The normal of dA_i is assumed
to be equal to the normal of the real surface in this point. The portion
of the energy landing on the finite surface and the energy radiated by the
differential surface element is called the **vertex-surface form factor** (or
vertex-patch form factor).

The vertex-surface form factor, based on equation 10.6, is:

$$F_{ij}^v = \frac{1}{dA_i} \cdot \int\limits_{dA_i} \int\limits_{\Delta A_j} H_{ij} \cdot \frac{dA_i \cdot \cos \phi_i \cdot dA_j \cdot \cos \phi_j}{\pi \cdot r^2} = \int\limits_{\Delta A_j} H_{ij} \cdot \frac{\cos \phi_i \cdot \cos \phi_j}{\pi \cdot r^2} \, dA_j.$$

(10.55)

This expression can either be evaluated by any discussed method or by
simply firing several rays from dA_i towards the centers of the patches gen-
erated by the subdivision of surface element ΔA_j. Each ray results in a
visibility factor of either 0 or 1, and an area-weighted summation has to be
carried out for those patches which have visibility 1 associated with them.

Suppose that in progressive refinement total and unshot radiosity esti-
mates are available for all vertices of surface elements. Unshot surface ra-
diosities can be approximated as the average of their unshot vertex radiosi-
ties. Having selected the surface element with the highest unshot radiosity
(U_j), and having also determined the vertex-surface form factors from all
the vertices to the selected surface (note that this is the reverse direction),
the new contributions to the total and unshot radiosities of vertices are:

$$\Delta B_i^v = \varrho_i \cdot U_j \cdot F_{ij}^v.$$

(10.56)

This has modified the total and unshot radiosities of the vertices. Thus,
estimating the surface radiosities, the last step can be repeated until con-
vergence, when the unshot radiosities of vertices become negligible. The
radiosity of the vertices can be directly turned to intensity and color in-
formation, enabling Gouraud's algorithm to complete the shading for the
internal pixels of the polygons.

10.3.2 Probabilistic progressive refinement

In probabilistic form factor computation, rays were fired from surfaces to
determine which other surfaces can absorb their radiosity. In progressive
refinement, on the other hand, the radiosity is shot proportionally to the

precomputed form factors. These approaches can be merged in a method which randomly shoots photons carrying a given portion of energy. As in progressive refinement, the unshot and total radiosities are initialized to the emission of the surfaces. At each step of the iteration a point is selected at random on the surface which has the highest unshot radiosity, a direction is generated according to the directional distribution of the radiation (cosine distribution), and a given portion, say $1/n$th, of the unshot energy is delivered to that surface which the photon encounters first on its way.

The program of this algorithm is then:

> **for** $j = 1$ **to** N **do** $B_j = U_j = E_j$
> **do**
> > j = Index of the surface of maximum U_j
> > $\vec{p}$ = a random point on surface j by uniform distribution
> > $\vec{d}$ = a random direction from $\vec{p}$ by cosine distribution
> > **if** $ray(\vec{p}, \vec{d})$ hits surface i first **then**
> > > U_i += $\varrho_i \cdot U_j/n$;
> > > B_i += $\varrho_i \cdot U_j/n$;
> > **endif**
> > U_j -= U_j/n;
> > $error = \max\{U_1, U_2, ..., U_N\}$;
> **while** $error > threshold$;

This is possibly the simplest algorithm for radiosity calculation. Since it does not rely on form factors, shading models other than diffuse reflection can also be incorporated.

10.4   Extensions to non-diffuse environments

The traditional radiosity methods discussed so far consider only diffuse reflections, having made it possible to ignore directional variation of the radiation of surfaces, since diffuse reflection generates the same radiant intensity in all directions. To extend the basic method taking into account more

terms in the general shading equation, directional dependence has to be built into the model.

The most obvious approach is to place a partitioned sphere on each elemental surface, and to calculate and store the intensity in each solid angle derived from the partition [ICG86]. This partitioning also transforms the integrals of the shading equations to finite sums, and limits the accuracy of the direction of the incoming light beams. Deriving a shading equation for each surface element and elemental solid angle, a linear equation is established, where the unknown variables are the radiant intensities of the surfaces in various solid angles. This linear equation can be solved by similar techniques to those discussed so far. The greatest disadvantage of this approach is that it increases the number of equations and the unknown variables by a factor of the number of partitioning solid angles, making the method prohibitively expensive.

More promising is the combination of the radiosity method with ray tracing, since the respective strong and weak points of the two methods tend to complement each other.

10.4.1   Combination of radiosity and ray tracing

In its simplest approach, the final, view-dependent step of the radiosity method involving Gouraud shading and usually z-buffering can be replaced by a recursive ray tracing algorithm, where the diffuse component is determined by the surface radiosities, instead of taking into consideration the abstract lightsources, while the surface radiosities are calculated by the methods we have discussed, ignoring all non-diffuse phenomena. The result is much better than the outcome of a simple recursive ray tracing, since the shadows lose their sharpness. The method still neglects some types of coupling, since, for example, it cannot consider the diffuse reflection of a light beam coherently reflected or refracted onto other surfaces. In figure 10.8, for example, the vase should have been illuminated by the light coherently reflected off the mirror, but the algorithm in question makes it dark, since the radiosity method ignores non-diffuse components. A possible solution to this problem is the introduction and usage of **extended form factors** [SP89].

A simplified shading model is used which breaks down the energy radiated by a surface into diffuse and coherently reflected or refracted components.
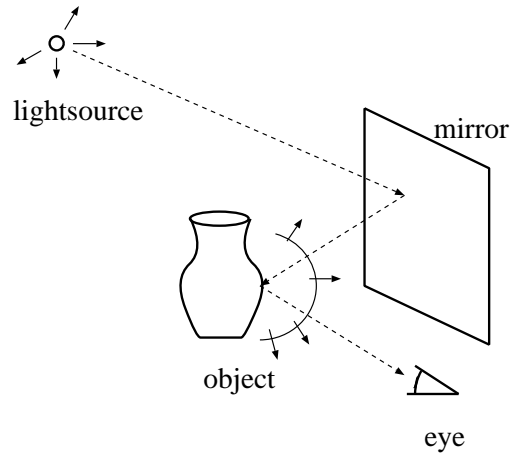
Figure 10.8: Coherent-diffuse coupling: The vase should have been illuminated by the light reflected off the mirror

An extended form factor F_{ij}^*, by definition represents that portion of the energy radiated diffusely by surface i which actually reaches surface j either by direct transmission or by single or multiple coherent reflections or refractions. The use of extended form factors allows for the calculation of the diffuse radiance of patches which takes into account not only diffuse but also coherent interreflections. Suppose diffuse radiance B_i of surface i needs to be calculated. Diffuse radiance B_i is determined by the diffuse radiation of other surfaces which reaches surface i and by those light components which are coherently reflected or refracted onto surface i. These coherent components can be broken down into diffuse radiances and emissions which are later coherently reflected or refracted several times, thus similar expression holds for the diffuse radiance in the improved model as for the original, only the normal form factors must be replaced by the extended ones. The extended radiosity equation defining the diffuse radiance of the surfaces in a non-diffuse environment is then:

$$B_i \cdot dA_i = E_i \cdot dA_i + \varrho_i \cdot \int B_j \cdot F_{ji}^* \cdot dA_j. \qquad (10.57)$$

Recursive ray tracing can be used to calculate the extended form factors. For each pixel of the hemicube a ray is generated which is traced backward

finding those surfaces which can be visited along this ray. For each surface found that portion of its diffusely radiated energy which reaches the previous surface along the ray should be computed — this is a differential form factor — then the attenuation of subsequent coherent reflections and refractions must be taken into consideration by multiplying the differential form factor by the product of the refractive and reflective coefficients of the surfaces visited by the ray between the diffuse source and surface i. Adding these portions of possible contribution for each pixel of the hemicube also taking the hemicube weighting function into account, the extended form factors can be generated. Having calculated the extended form factors, the radiosity equation can be solved by the method discussed, resulting in the diffuse radiosities of the surfaces.

Expensive ray-tracing can be avoided and normal form factors can be worked with if only single, ideal, mirror-like coherent reflections are allowed, because this case can be supported by mirroring every single surface onto the reflective surfaces. We can treat these reflective surfaces as windows onto a "mirror world", and the normal form factor between the mirrored surface and another surface will be responsible for representing that part of energy transfer which would be represented by the difference of the extended and normal form factors [WCG87].

If the diffuse radiosities of the surfaces are generated, then in the second, view-dependent phase another recursive ray-tracing algorithm can be applied to generate the picture. Whenever a diffuse intensity is needed this second pass ray-tracing will use the radiosities computed in the first pass. In contrast to the naive combination of ray-tracing and radiosity, the diffuse radiosities are now correct, since the first pass took not only the diffuse interreflections but also the coherent interreflections and refractions into consideration.

10.5 Higher order radiosity approximation

The original radiosity method is based on finite element techniques. In other words, the radiosity distribution is searched in a piecewise constant function form, reducing the original problem to the calculation of the values of the steps.

The idea of piecewise constant approximation is theoretically simple and easy to accomplish, but an accurate solution would require a large number of steps, making the solution of the linear equation difficult. Besides, the constant approximation can introduce unexpected artifacts in the picture even if it is softened by Gouraud shading.

This section addresses this problem by applying a variational method for the solution of the integral equation [SK93].

The variational solution consists of the following steps [Mih70]:

1. It establishes a functional which is extreme for a function (radiosity distribution) if and only if the function satisfies the original integral equation (the basic radiosity equation).

2. It generates the extreme solution of the functional by **Ritz's method**, that is, it approximates the function to be found by a function series, where the coefficients are unknown parameters, and the extremum is calculated by making the partial derivatives of the functional (which is a function of the unknown coefficients) equal to zero. This results in a linear equation which is solved for the coefficients defining the radiosity distribution function.

Note the similarities between the second step and the original radiosity method. The proposed variational method can, in fact, be regarded as a generalization of the finite element method, and, as we shall see, it contains that method if the basis functions of the function series are selected as piecewise constant functions being equal to zero except for a small portion of the surfaces. Nevertheless, we are not restricted to these basis functions, and can select other function bases, which can approximate the radiosity distribution more accurately and by fewer basis functions, resulting in a better solution and requiring the calculation of a significantly smaller linear equation.

Let the diffuse coefficient be $\varrho(p)$ at point p and the visibility indicator between points p and p' be $H(p, p')$. Using the notations of figure 10.9, and denoting the radiosity and emission at point p by $B(p)$ and $E(p)$ respectively, the basic radiosity equation is:

$$B(p) \cdot dA = E(p) \cdot dA + \varrho(p) \cdot \int_A B(p') \, f(p, p') \, dA' \cdot dA \qquad (10.58)$$
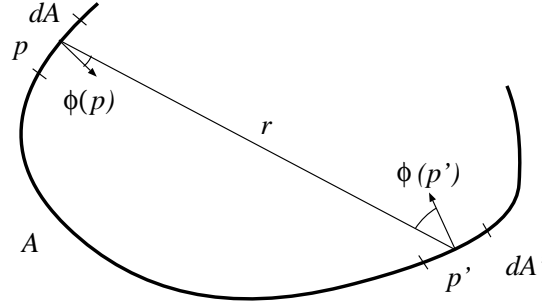
Figure 10.9: Geometry of the radiosity calculation

where $f(p, p')$ is the **point-to-point form factor**:

$$f(p, p') = H(p, p') \, \frac{\cos \phi(p) \cdot \cos \phi(p')}{r^2 \pi}. \tag{10.59}$$

Dividing both sides by dA, the **radiosity equation** is then:

$$B(p) = E(p) + \varrho(p) \cdot \int_A B(p') \, f(p, p') \, dA'. \tag{10.60}$$

Let us define a linear operator $\mathcal{L}$:

$$\mathcal{L}B(p) = B(p) - \varrho(p) \cdot \int_A B(p') \, f(p, p') \, dA'. \tag{10.61}$$

Then the radiosity equation can also be written as follows:

$$\mathcal{L}B(p) = E(p). \tag{10.62}$$

The solution of the radiosity problem means to find a function B satisfying this equation. The domain of possible functions can obviously be restricted to functions whose square has finite integration over surface A. This function space is usually called $L_2(A)$ space where the scalar product is defined as:

$$\langle u, v \rangle = \int_A u(p) \cdot v(p) \, dA. \tag{10.63}$$

If $\mathcal{L}$ were a symmetric and positive operator, that is, for any u, v in $L_2(A)$,

$$\langle \mathcal{L}u, v \rangle = \langle u, \mathcal{L}v \rangle \tag{10.64}$$

were an identity and

$$\langle \mathcal{L}u, u \rangle \geq 0 \qquad \wedge \qquad \langle \mathcal{L}u, u \rangle = 0 \quad \text{if and only if } u = 0, \tag{10.65}$$

then according to the *minimal theorem of quadratic functionals* [Ode76] the solution of equation 10.62 could also be found as the stationary point of the following functional:

$$\langle \mathcal{L}B, B \rangle - 2\langle E, B \rangle + \langle E, E \rangle. \tag{10.66}$$

Note that $\langle E, E \rangle$ makes no difference in the stationary point, since it does not depend on B, but it simplifies the resulting formula.

To prove that if and only if some B_0 satisfies

$$\mathcal{L}B_0 = E \tag{10.67}$$

for a symmetric and positive operator $\mathcal{L}$, then B_0 is extreme for the functional of equation 10.66, a sequence of identity relations based on the assumption that $\mathcal{L}$ is positive and symmetric can be used:

$$\langle \mathcal{L}B, B \rangle - 2\langle E, B \rangle + \langle E, E \rangle = \langle \mathcal{L}B, B \rangle - 2\langle \mathcal{L}B_0, B \rangle + \langle E, E \rangle =$$

$$\langle \mathcal{L}B, B \rangle - \langle \mathcal{L}B_0, B \rangle - \langle B_0, \mathcal{L}B \rangle + \langle E, E \rangle =$$

$$\langle \mathcal{L}B, B \rangle - \langle \mathcal{L}B_0, B \rangle - \langle \mathcal{L}B, B_0 \rangle + \langle \mathcal{L}B_0, B_0 \rangle - \langle \mathcal{L}B_0, B_0 \rangle + \langle E, E \rangle =$$

$$\langle \mathcal{L}(B - B_0), (B - B_0) \rangle - \langle \mathcal{L}B_0, B_0 \rangle + \langle E, E \rangle. \tag{10.68}$$

Since only the term $\langle \mathcal{L}(B - B_0), (B - B_0) \rangle$ depends on B and this term is minimal if and only if $B - B_0$ is zero due to the assumption that $\mathcal{L}$ is positive, therefore the functional is really extreme for that B_0 which satisfies equation 10.62.

Unfortunately $\mathcal{L}$ is not symmetric in its original form (equation 10.61) due to the asymmetry of the radiosity equation which depends on $\varrho(p)$ but not on $\varrho(p')$. One possible approach to this problem is the subdivision of surfaces into finite patches having constant diffuse coefficients, and working

with multi-variate functionals, but this results in a significant computational overhead.

Now another solution is proposed that eliminates the asymmetry by calculating $B(p)$ indirectly through the generation of $B(p)/\sqrt{\varrho(p)}$. In order to do this, both sides of the radiosity equation are divided by $\sqrt{\varrho(p)}$:

$$\frac{E(p)}{\sqrt{\varrho(p)}} = \frac{B(p)}{\sqrt{\varrho(p)}} - \sqrt{\varrho(p)} \int_A \frac{B(p')}{\sqrt{\varrho(p')}} \sqrt{\varrho(p')} \, f(p,p') \, dA'. \qquad (10.69)$$

Let us define $B^*(p)$, $E^*(p)$ and $g(p,p')$ by the following formulae:

$$B^*(p) = \frac{B(p)}{\sqrt{\varrho(p)}}, \qquad E^*(p) = \frac{E(p)}{\sqrt{\varrho(p)}}, \qquad g(p,p') = f(p,p')\sqrt{\varrho(p)\varrho(p')}.$$

$$(10.70)$$

Using these definitions, we get the following form of the original radiosity equation:

$$E^*(p) = B^*(p) - \int_A B^*(p') \, g(p,p') \, dA'. \qquad (10.71)$$

Since $g(p,p') = g(p',p)$, this integral equation is defined by a symmetric linear operator $\mathcal{L}^*$:

$$\mathcal{L}^* B^*(p) = B^*(p) - \int_A B^*(p') \, g(p,p') \, dA'. \qquad (10.72)$$

As can easily be proven, operator $\mathcal{L}^*$ is not only symmetric but also positive taking into account that for physically correct models:

$$\int_A \int_A B(p')g(p,p') \, dA'dA \leq \int_A B(p) \, dA. \qquad (10.73)$$

This means that the solution of the modified radiosity equation is equivalent to finding the stationary point of the following functional:

$$I(B^*) = \langle \mathcal{L}^* B^*, B^* \rangle - 2\langle E^*, B^* \rangle + \langle E^*, E^* \rangle =$$

$$\int_A (E^*(p) - B^*(p))^2 dA - \int_A \int_A B^*(p) \, B^*(p') \, g(p,p') \, dA \, dA'. \qquad (10.74)$$

This extreme property of functional I can also be proven by generating the functional's first variation and making it equal to zero:

$$0 = \delta I = \frac{\partial I(B^* + \alpha \, \delta B)}{\partial \alpha}\Big|_{\alpha=0}. \qquad (10.75)$$

Using elementary derivation rules and taking into account the following symmetry relation:

$$\int_A \int_A B^*(p)\ \delta B(p')\ g(p,p')\ dAdA' = \int_A \int_A \delta B(p)\ B^*(p')\ g(p,p')\ dAdA'$$

$$(10.76)$$

the formula of the first variation is transformed to:

$$0 = \delta I = \int_A [E^*(p) - B^*(p) + \int_A B^*(p') \cdot g(p,p')\ dA'] \cdot \delta B\ dA. \quad (10.77)$$

The term closed in brackets should be zero to make the expression zero for any δB variation. That is exactly the original radiosity equation, hence finding the stationary point of functional I is really equivalent to solving integral equation 10.71.

In order to find the extremum of functional $I(B^*)$, Ritz's method is used. Assume that the unknown function B^* is approximated by a function series:

$$B^*(p) \approx \sum_{k=1}^{n} a_k \cdot b_k(p) \qquad (10.78)$$

where $(b_1, b_2, ..., b_n)$ form a complete function system (that is, any piecewise continuous function can be approximated by their linear combination), and $(a_1, a_2, ..., a_n)$ are unknown coefficients. This assumption makes functional $I(B^*)$ an n-variate function $I(a_1, ..., a_n)$, which is extreme if all the partial derivatives are zero. Having made every $\partial I/\partial a_k$ equal to zero, a linear equation system can be derived for the unknown a_k-s ($k = \{1, 2, ..., n\}$):

$$\sum_{i=0}^{n} a_i [\int_A b_i(p)b_k(p)dA - \int_A \int_A b_k(p)b_i(p')g(p,p')\ dAdA'] = \int_A E^*(p)b_k(p)dA.$$

$$(10.79)$$

This general formula provides a linear equation for any kind of complete function system $b_1, ..., b_n$, thus it can be regarded as a basis of many different radiosity approximation techniques, because the different selection of basis functions, b_i, results in different methods of determining the radiosity distribution.

Three types of function bases are discussed:

- piecewise constant functions which lead to the traditional method, proving that the original approach is a special case of this general framework,

- piecewise linear functions which, as we shall see, are not more difficult than the piecewise constant approximations, but they can provide more accurate solutions. It is, in fact, a refined version of the method of "**vertex-surface form factors**",

- harmonic (cosine) functions where the basis functions are not of finite element type because they can approximate the radiosity distribution everywhere not just in a restricted part of the domain, and thus fall into the category of global element methods.
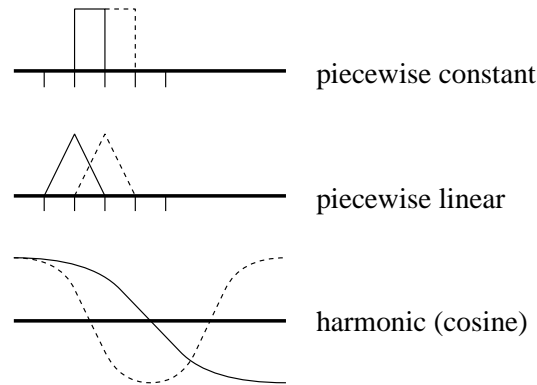
Figure 10.10: One-dimensional analogy of proposed basis functions

10.5.1 Piecewise constant radiosity approximation

Following a finite element approach, an appropriate set of b_k functions can be defined having broken down the surface into ΔA_1, ΔA_2,...,ΔA_n surface elements:

$$b_k(p) = \begin{cases} 1 \text{ if } p \text{ is on } \Delta A_k \\ \\ 0 \text{ otherwise} \end{cases} \tag{10.80}$$

If the emission E and the diffuse coefficient ϱ are assumed to be constant on the elemental surface ΔA_k and equal to E_k and ϱ_k respectively, equation 10.79 will have the following form:

$$a_k \Delta A_k - \sum_{i=0}^{n} a_i [\int_{\Delta A_k} \int_{\Delta A_i} g(p, p') \, dA dA'] = \frac{E_k}{\sqrt{\varrho_k}} \Delta A_k. \qquad (10.81)$$

According to the definition of basis function b_k, the radiosity of patch k is:

$$B_k = B_k^* \sqrt{\varrho_k} = a_k \sqrt{\varrho_k}. \qquad (10.82)$$

Substituting this into equation 10.81 and using the definition of $g(p, p')$ in equation 10.70, we get:

$$B_k \Delta A_k - \varrho_k \sum_{i=0}^{n} B_i [\int_{\Delta A_k} \int_{\Delta A_i} f(p, p') \, dA dA'] = E_k \Delta A_k. \qquad (10.83)$$

Let us introduce the **patch-to-patch form factor** as follows:

$$F_{ki} = \frac{1}{\Delta A_k} \int_{\Delta A_k} \int_{\Delta A_i} f(p, p') \, dA dA'. \qquad (10.84)$$

Note that this is the usual definition taking into account the interpretation of $f(p, p')$ in equation 10.59.

Dividing both sides by ΔA_k, the linear equation is then:

$$B_k - \varrho_k \sum_{i=0}^{n} B_i F_{ki} = E_k. \qquad (10.85)$$

This is exactly the well known linear equation of original radiosity method (equation 10.10). Now let us begin to discuss how to define and use other, more effective function bases.

10.5.2   Linear finite element techniques

Let us decompose the surface into planar triangles and assume that the radiosity variation is linear on these triangles. Thus, each vertex i of the triangle mesh will correspond to a "tent shaped" basis function b_i that is 1

at this vertex and linearly decreases to 0 on the triangles incident to this vertex.

Placing the center of the coordinate system into vertex i, the position vector of points on an incident triangle can be expressed by a linear combination of the edge vectors $\vec{a}, \vec{b}$:

$$\vec{p} = \alpha\vec{a} + \beta\vec{b} \tag{10.86}$$

with $\alpha, \beta \geq 0 \quad \wedge \quad \alpha + \beta \leq 1$.
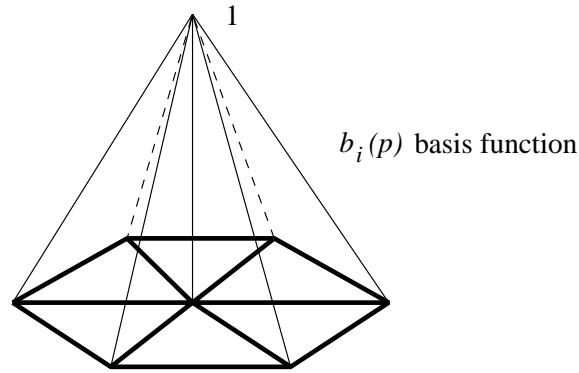
Figure 10.11: Linear basis function in three dimensions

Thus, the surface integral of some function F on a triangle can be written as follows:

$$\int\limits_{\Delta A} F(\vec{p})dA = \int\limits_{\beta=0}^{1} \int\limits_{\alpha=0}^{1-\beta} F(\alpha, \beta)|\vec{a} \times \vec{b}|d\alpha d\beta =$$

$$2\Delta A \int\limits_{\beta=0}^{1} \int\limits_{\alpha=0}^{1-\beta} F(\alpha, \beta) \, d\alpha d\beta. \tag{10.87}$$

If $F(\alpha, \beta)$ is a polynomial function, then its surface integration can be determined in closed form by this formula.

The basis function which is linearly decreasing on the triangles can be conveniently expressed by α, β coordinates:

$$
\begin{aligned}
b_k(\alpha, \beta) &= 1 - \alpha - \beta, \\
b_{k'}(\alpha, \beta) &= \alpha, \\
b_{k''}(\alpha, \beta) &= \beta, \\
b_i &= 0 \qquad\qquad \text{if } i \neq k, k', k''
\end{aligned}
\tag{10.88}
$$

where k, k' and k'' are the three vertices of the triangle.

Let us consider the general equation (equation 10.79) defining the weights of basis functions; that is the radiosities at triangle vertices for linear finite elements. Although its integrals can be evaluated directly, it is worth examining whether further simplification is possible. Equation 10.79 can also be written as follows:

$$
\int_A [\sum_{i=0}^n a_i \{ b_i(p) - \int_A b_i(p')g(p,p')\ dA' \} - E^*(p)] \cdot b_k(p)\ dA = 0
\tag{10.89}
$$

The term enclosed in brackets is a piecewise linear expression according to our assumption if E^* is also linear. The integration of the product of this expression and any linear basis function is zero. That is possible if the term in brackets is constantly zero, thus an equivalent system of linear equations can be derived by requiring the closed term to be zero in each vertex k (this implies that the function will be zero everywhere because of linearity):

$$
a_k - \sum_{i=0}^n a_i \int_A b_i(p')g(p_k,p')\ dA' = E_k^*, \qquad k = \{1, 2, ...n\}
\tag{10.90}
$$

As in the case of piecewise constant approximation, the diffuse coefficient $\varrho(p)$ is assumed to be equal to ϱ_k at vertex k, and using the definitions of the normalized radiosities we can conclude that:

$$
a_k = B_k^* = \frac{B_k}{\sqrt{\varrho_k}}, \qquad E_k^* = \frac{E_k}{\sqrt{\varrho_k}}.
\tag{10.91}
$$

Substituting this into equation 10.90 and taking into account that b_i is zero outside ΔA_i, we get:

$$
B_k - \varrho_k \sum_{i=0}^n B_i[\int_{\Delta A_i} b_i(p')f(p_k,p')\sqrt{\frac{\varrho(p')}{\varrho_i}}\ dA'] = E_k.
\tag{10.92}
$$

Let us introduce the **vertex-patch form factor** P_{ki}:

$$P_{ki} = \int\limits_{\Delta A_i} b_i(p')f(p_k,p')\sqrt{\frac{\varrho(p')}{\varrho_i}}\; dA'. \qquad (10.93)$$

If the diffuse coefficient can be assumed to be (approximately) constant on the triangles adjacent to vertex i, then:

$$P_{ki} \approx \int\limits_{\Delta A_i} b_i(p')f(p_k,p')\; dA'. \qquad (10.94)$$

The linear equation of the vertex radiosities is then:

$$B_k - \varrho_k \sum_{i=0}^{n} B_i P_{ki} = E_k. \qquad (10.95)$$

This is almost the same as the linear equation describing the piecewise constant approximation (equation 10.85), except that:

- Unknown parameters $B_1, ..., B_k$ represent now vertex radiosities rather than patch radiosities. According to Euler's law, the number of vertices of a triangular faced polyhedron is half of the number of its faces plus two. Thus the size of the linear equation is almost the same as for the number of quadrilaterals used in the original method.

- There is no need for double integration and thus the linear approximation requires a simpler numerical integration to calculate the form factors than constant approximation.

The vertex-patch form factor can be evaluated by the techniques developed for patch-to-patch form factors taking account also the linear variation due to b_i. This integration can be avoided, however, if linear approximation of $f(p_k, p')$ is acceptable. One way of achieving this is to select the subdivision criterion of surfaces into triangles accordingly.

A linear approximation can be based on point-to-point form factors between vertex k and the vertices of triangle $\Delta A'$. Let the $f(p_k, p)$ values of the possible combinations of point p_k and the vertices be F_1, F_2, F_3 respectively. A linear interpolation of the point-to-point form factor between p_k and $p' = \alpha'\vec{a}' + \beta'\vec{b}'$ is:

$$f(p_k, p') = \alpha'F_1 + \beta'F_2 + (1 - \alpha' - \beta')F_3. \qquad (10.96)$$

Using this assumption the surface integral defining P_{ki} can be expressed in closed form.

10.5.3 Global element approach — harmonic functions

In contrast to previous cases, the application of harmonic functions does not require the subdivision of surfaces into planar polygons, but deals with the original geometry. This property makes it especially useful when the view-dependent rendering phase uses ray-tracing.

Suppose surface A is defined parametrically by a position vector function, $\vec{r}(u, v)$, where parameters u and v are in the range of $[0, 1]$.

Let a representative of the basis functions be:

$$b_{ij} = \cos(i\pi u) \cdot \cos(j\pi v) = C_u^i C_v^j \qquad (10.97)$$

(C_u^i substitutes $\cos(i\pi u)$ for notational simplicity). Note that the basis functions have two indices, hence the sums should also be replaced by double summation in equation 10.79. Examining the basis functions carefully, we can see that the goal is the calculation of the Fourier series of the radiosity distribution.

In contrast to the finite element method, the basis functions are now non-zero almost everywhere in the domain, so they can approximate the radiosity distribution in a wider range. For that reason, approaches applying this kind of basis function are called **global element methods**.

In the radiosity method the most time consuming step is the evaluation of the integrals appearing as coefficients of the linear equation system (equation 10.79). By the application of cosine functions, however, the computational time can be reduced significantly, because of the orthogonal properties of the trigonometric functions, and also by taking advantage of effective algorithms, such as Fast Fourier Transform (FFT).

In order to illustrate the idea, the calculation of

$$\int_A E^*(p) b_{kl}(p) \, dA$$

for each k, l is discussed. Since $E^*(p) = E^*(\vec{r}(u, v))$, it can be regarded as a function defined over the square $[0, 1]^2$. Using the equalities of surface

integrals, and introducing the notation $J(u,v) = |\partial \vec{r}/\partial u \times \partial \vec{r}/\partial v|$ for surface element magnification, we get:

$$\int_A E^*(p) b_{kl}(p) \, dA = \int_0^1 \int_0^1 E^*(\vec{r}(u,v)) b_{kl}(u,v) J(u,v) \, du dv. \qquad (10.98)$$

Let us mirror the function $E^*(\vec{r}) \cdot J(u,v)$ onto coordinate system axes u and v, and repeat the resulting function having its domain in $[-1,1]^2$ infinitely in both directions with period 2. Due to mirroring and periodic repetition, the final function $\hat{E}(u,v)$ will be even and periodic with period 2 in both directions. According to the theory of the Fourier series, the function can be approximated by the following sum:

$$\hat{E}(u,v) \approx \sum_{i=0}^m \sum_{j=0}^m E_{ij} C_u^i C_v^j. \qquad (10.99)$$

All the Fourier coefficients E_{ij} can be calculated by a single, two-dimensional FFT. (A D-dimensional FFT of N samples can be computed by taking DN^{D-1} number of one-dimensional FFTs [Nus82] [PFTV88].)

Since $\hat{E}(u,v) = E^*(\vec{r}) \cdot J(u,v)$ if $0 \le u,v \le 1$, this Fourier series and the definition of the basis functions can be applied to equation 10.98, resulting in:

$$\int_A E^*(p) b_{kl}(p) \, dA = \int_{u=0}^1 \int_{v=0}^1 \sum_{i=0}^m \sum_{j=0}^m E_{ij} C_u^i C_v^j \cdot b_{kl}(u,v) \, du dv =$$

$$\sum_{i=0}^m \sum_{j=0}^m E_{ij} \int_0^1 C_u^i C_u^k du \int_0^1 C_v^j C_v^l dv = \begin{cases} E_{0,0} & \text{if } k = 0 \text{ and } l = 0 \\[2mm] E_{0,l}/2 & \text{if } k = 0 \text{ and } l \ne 0 \\[2mm] E_{k,0}/2 & \text{if } k \ne 0 \text{ and } l = 0 \\[2mm] E_{k,l}/4 & \text{if } k \ne 0 \text{ and } l \ne 0 \end{cases} \qquad (10.100)$$

Consequently, the integral can be calculated in closed form, having replaced the original function by Fourier series. Similar methods can be used to evaluate the other integrals. In order to compute

$$\int_A b_{ij}(p) b_{kl}(p) dA$$

$J(u,v)$ must be Fast Fourier Transformed.

To calculate

$$\int\limits_{A} \int\limits_{A} b_k(p)b_i'(p)g(p,p')\ dAdA'$$

the Fourier transform of

$$g(p(u,v),p'(u',v')) \cdot J(u,v)J(u',v')$$

is needed. Unfortunately the latter requires a 4D FFT which involves many operations. Nevertheless, this transform can be realized by two two-dimensional FFTs if $g(p,p')$ can be assumed to be nearly independent of either p or p', or it can be approximated by a product form of p and p' independent functions.

Finally, it should be mentioned that other **global function bases** can also be useful. For example, Chebyshev polynomials are effective in approximation, and similar techniques to FFT can be developed for their computation.

Chapter 11

SAMPLING AND QUANTIZATION ARTIFACTS

From the information or signal processing point of view, modeling can be regarded as the definition of the intended world by digital and discrete data which are processed later by image synthesis. Since the intended world model, like the real world, is continuous, modeling always involves an analog-digital conversion to the internal representation of the digital computer. Later in image synthesis, the digital model is resampled and re-quantized to meet the requirements of the display hardware, which is much more drastic than the sampling of modeling, making this step responsible for the generation of artifacts due to the approximation error in the sampling process. In this chapter, the problems of discrete sampling will be discussed first, then the issue of quantization will be addressed.

The sampling of a two-dimensional color distribution, $I(x, y)$, can be described mathematically as a multiplication by a "**comb function**" which keeps the value of the sampled function in the sampling points, but makes it zero elsewhere:

$$I_s(x, y) = I(x, y) \cdot \sum_i \sum_j \delta(x - i \cdot \Delta x, y - j \cdot \Delta y). \qquad (11.1)$$

The 2D Fourier transformation of this signal is:

$$I_s^*(\alpha, \beta) = \int\limits_{-\infty}^{\infty} \int\limits_{-\infty}^{\infty} I_s(x, y) \cdot e^{-jx\alpha} \cdot e^{-jy\beta} dx dy =$$

$$\frac{1}{\Delta x \cdot \Delta y} \sum_i \sum_j I^*(\alpha - \frac{2\pi i}{\Delta x}, \beta - \frac{2\pi j}{\Delta y}). \tag{11.2}$$

The sampling would be correct if the requirements of the **sampling theorem** could be met. The sampling theorem states that a continuous signal can be reconstructed exactly from its samples by an ideal low-pass filter only if it is band-limited to a maximum frequency which is less than half of the sampling frequency. That is also obvious from equation 11.2, since it repeats the spectrum of the signal infinitely by periods $2\pi/\Delta x$ and $2\pi/\Delta y$, which means that the spectrum of non-band-limited signals will be destroyed by their repeated copies.
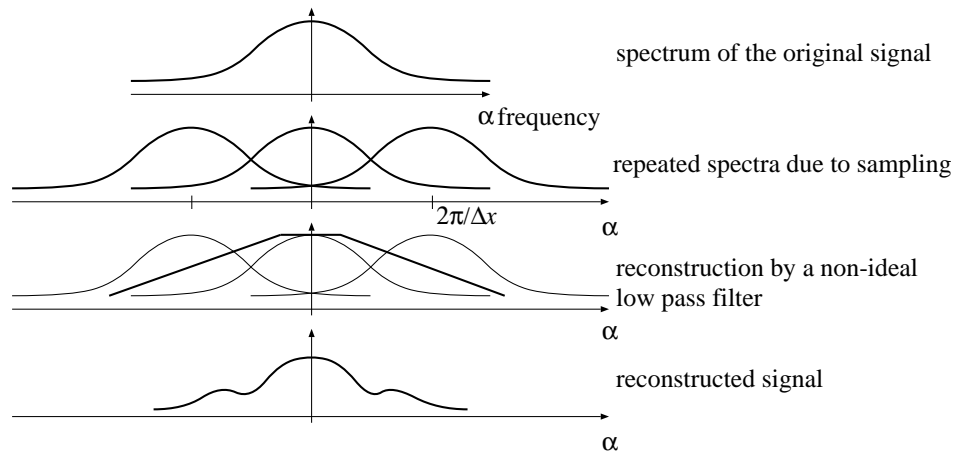
Figure 11.1: Analysis of the spectrum of the sampled color distribution

The real world is never band-limited, because objects appear suddenly (like step functions) as we move along a path, introducing infinitely high frequencies. Thus, the sampling theorem can never be satisfied in computer graphics, causing the repeated spectra of the color distribution to overlap, and destroying even the low frequency components of the final spectrum.

The phenomenon of the appearance of high frequency components in the lower frequency ranges due to incorrect sampling is called **aliasing** (figure 11.1). The situation is made even worse by the method of reconstruction of the continuous signal. As has been discussed, raster systems use a 0-order hold circuit in their D/A converter to generate a continuous signal, which is far from being an ideal low-pass filter.

The results of the unsatisfactory sampling and reconstruction are well-known in computer graphics. Polygon edges will have stairsteps or jaggies which are aliases of unwanted high frequency components. The sharp corners of the jaggies are caused by the inadequate low-pass filter not suppressing those higher components. The situation is even worse for small objects of a size comparable with pixels, such as small characters or textures, because they can totally disappear from the screen depending on the actual sampling grid. In order to reduce the irritating effects of aliasing, three different approaches can be taken:

1. Increasing the resolution of the displays. This approach has not only clear technological constraints, but has proven inefficient for eliminating the effects of aliasing, since the human eye is very sensitive to the regular patterns that aliasing causes.

2. Band-limiting the image by applying a low pass filter before sampling. Although the high frequency behavior of the image will not be accurate, at least the more important low frequency range will not be destroyed by aliases. This **anti-aliasing** approach is called **pre-filtering**, since the filtering is done before the sampling.

3. The method which filters the generated image after sampling is called **post-filtering**. Since the signal being sampled is not band-limited, the aliases will inevitably occur on the image, and cannot be removed by a late filtering process. If the sampling uses the resolution of the final image, the same aliases occur, and post-filtering can only reduce the sharp edges of jaggies, improving the reconstruction process. The filtering cannot make a distinction between aliases and normal image patterns, causing a decrease in the sharpness of the picture. Thus, post-filtering is only effective if it is combined with higher resolution sampling, called **supersampling**, because the higher sampling frequency will reduce the inevitable aliasing if the spectrum energy

falls off with increasing frequency, since higher sampling frequency increases the periods of repetition of the spectrum by factors $2\pi/\Delta x$ and $2\pi/\Delta y$. Supersampling generates the image at a higher resolution than is needed by the display hardware, the final image is then produced by sophisticated digital filtering methods which produce the filtered image at the required resolution.

Comparing the last two basic approaches, we can conclude that prefiltering works in continuous space allowing for the elimination of aliases in theory, but that efficiency considerations usually inhibit the use of accurate low-pass filters in practice. Post-filtering, on the other hand, samples the non-band-limited signal at a higher sampling rate and reduces, but does not eliminate aliasing, and allows a fairly sophisticated and accurate low-pass filter to be used for the reconstruction of the continuous image at the normal resolution.

11.1   Low-pass filtering

According to the **sampling theorem**, the **Nyquist limits** of a 2D signal sampled at $\Delta x, \Delta y$ periodicity are $\pi/\Delta x$ and $\pi/\Delta y$ respectively, requiring a low-pass filter suppressing all frequency components above the Nyquist limits, but leaving those below the cut-off point untouched. The filter function in the frequency domain is:

$$F(\alpha, \beta) = \begin{cases} 1 \text{ if } |\alpha| < \pi/\Delta x \text{ and } |\beta| < \pi/\Delta y \\ \\ 0 \text{ otherwise} \end{cases} \tag{11.3}$$

The filtering process is a multiplication by the filter function in the frequency domain, or equivalently, a convolution in the spatial domain by the pulse response of the filter ($f(x,y)$) which is the inverse Fourier transform of the filter function:

$$I_f^*(\alpha, \beta) = I^*(\alpha, \beta) \cdot F(\alpha, \beta),$$

$$I_f(x,y) = I(x,y) * f(x,y) = \int\limits_{-\infty}^{\infty} \int\limits_{-\infty}^{\infty} I(t, \tau) \cdot f(x - t, y - \tau) \, dt d\tau. \tag{11.4}$$

The pulse response of the ideal low-pass filter is based on the well-known sinc function:

$$f(x,y) = \frac{\sin(x \cdot \pi/\Delta x)}{x \cdot \pi/\Delta x} \cdot \frac{\sin(y \cdot \pi/\Delta y)}{y \cdot \pi/\Delta y} = \text{sinc}(\frac{x \cdot \pi}{\Delta x}) \cdot \text{sinc}(\frac{y \cdot \pi}{\Delta y}). \quad (11.5)$$

The realization of the low-pass filtering as a convolution with the 2D sinc function has some serious disadvantages. The sinc function decays very slowly, thus a great portion of the image can affect the color on a single point of the filtered picture, making the filtering complicated to accomplish. In addition to that, the sinc function has negative portions, which may result in negative colors in the filtered image. Sharp transitions of color in the original image cause noticeable ringing, called the Gibbs phenomenon, in the filtered picture. In order to overcome these problems, all positive, smooth, finite extent or nearly finite extent approximations of the ideal sinc function are used for filtering instead. These filters are expected to have unit gain for $\alpha = 0, \beta = 0$ frequencies, thus the integral of their impulse response must also be 1:

$$F(0,0) = \int\limits_{-\infty}^{\infty} \int\limits_{-\infty}^{\infty} f(x,y) \, dxdy = 1 \qquad (11.6)$$

Some of the most widely used filters (figure 11.2) are:

1. **Box filter**: In the spatial domain:

$$f(x,y) = \begin{cases} 1 \text{ if } |x| < \Delta x/2 \text{ and } |y| < \Delta y/2 \\ \\ 0 \text{ otherwise} \end{cases} \qquad (11.7)$$

   In the frequency domain the box filter is a sinc function which is not at all accurate approximation of the ideal low-pass filters.

2. **Cone filter**: In the spatial domain, letting the normalized distance from the point (0,0) be $r(x,y) = \sqrt{(x/\Delta x)^2 + (y/\Delta y)^2}$:

$$f(x,y) = \begin{cases} (1 - r) \cdot 3/\pi \text{ if } r < 1 \\ \\ 0 \text{ otherwise} \end{cases} \qquad (11.8)$$
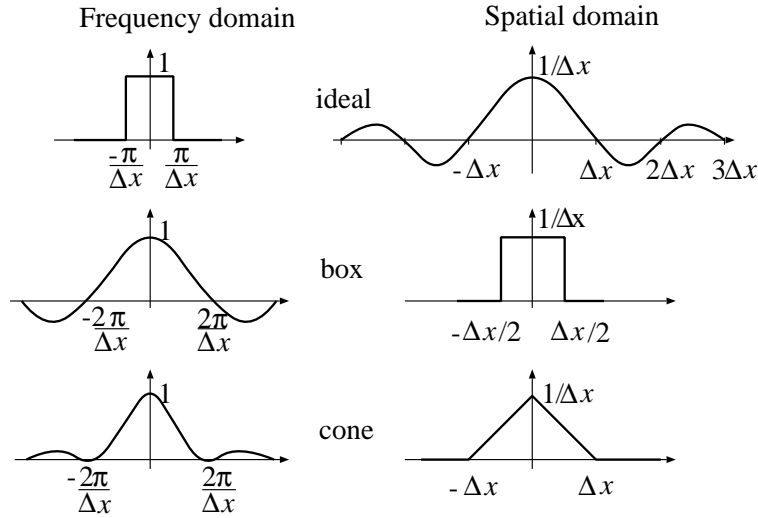
Figure 11.2: Frequency and spatial behavior of ideal and approximate low-pass filters

The coefficient $3/\pi$ guarantees that the total volume of the cone, that is the integral of the impulse response of the filter, is 1. The Fourier transformation of this impulse response is a sinc^2 type function which provides better high frequency suppression than the box filter.

3. **Gaussian filter**: This filter uses the Gaussian distribution function e^{-r^2} to approximate the sinc function by a positive, smooth function, where $r = \sqrt{(x/\Delta x)^2 + (y/\Delta y)^2}$ as for the cone filter. Although the Gaussian is not a finite extent function, it decays quickly making the contribution of distant points negligible.

Having defined the filter either in the frequency or in the spatial domain, there are basically two ways to accomplish the filtering. It can be done either in the spatial domain by evaluating the convolution of the original image and the impulse response of the filter, or in the frequency domain by multiplying the frequency distributions of the image by the filter function. Since the original image is available in spatial coordinates, and the filtered image is also expected in spatial coordinates, the latter approach requires a transformation of the image to the frequency domain before the filtering,

then a transformation back to the spatial domain after the filtering. The computational burden of the two Fourier transformations makes frequency domain filtering acceptable only for special applications, even if effective methods, such as Fast Fourier Transform (FFT), are used.

11.2 Pre-filtering anti-aliasing techniques

Pre-filtering methods sample the image after filtering at the sample rate defined by the resolution of the display hardware ($\Delta x = 1, \Delta y = 1$). If the filtering has been accomplished in the spatial domain, then the filtered and sampled signal is:

$$I_{sf}(x,y) = [I(x,y) * f(x,y)] \cdot \sum_i \sum_j \delta(x - i, y - j). \qquad (11.9)$$

For a given pixel of X, Y integer coordinates:

$$I_{sf}(X,Y) = I(x,y) * f(x,y) \cdot \delta(x-X, y-Y) = \int\limits_{-\infty}^{\infty} \int\limits_{-\infty}^{\infty} I(t,\tau) \cdot f(X-t, Y-\tau) \, dt d\tau.$$
$$(11.10)$$

For finite extent impulse response (FIR) filters, the infinite range of the above integral is replaced by a finite interval. For a box filter:

$$I_{s,\text{box}}(X,Y) = \int\limits_{X-0.5}^{X+0.5} \int\limits_{Y-0.5}^{Y+0.5} I(t,\tau) \, dt d\tau. \qquad (11.11)$$

Suppose P number of constant color primitives have intersection with the 1×1 rectangle of X, Y pixel. Let the color and the area of intersection of a primitive p be I_p and A_p, respectively. The integral 11.11 is then:

$$I_{s,\text{box}}(X,Y) = \sum_{p=1}^{P} I_p \cdot A_p. \qquad (11.12)$$

For a cone filter, assuming a polar coordinate system (r, ϕ) centered around (X, Y), the filtered signal is:

$$I_{s,\text{cone}}(X,Y) = \frac{3}{\pi} \int\limits_{x^2+y^2 \leq 1} \int I(x,y) \cdot (1-r) \, dx dy = \frac{3}{\pi} \int\limits_{r=0}^{1} \int\limits_{\phi=0}^{2\pi} I(r,\phi) \cdot (1-r) \cdot r \, d\phi dr.$$
$$(11.13)$$

As for the box filter, the special case is examined when P constant color primitives can contribute to a pixel, that is, they have intersection with the unit radius circle around the (X, Y) pixel, assuming the color and the area of intersection of primitive p to be I_p and A_p, respectively:

$$I_{s,\text{cone}}(X, Y) = \frac{3}{\pi} \sum_{p=1}^{P} I_p \int\limits_{A_p} (1 - r) \, dA \qquad (11.14)$$

where $\int\limits_{A_p} (1 - r) \, dA$ is the volume above the A_p area bounded by the surface of the cone.

11.2.1 Pre-filtering regions

An algorithm which uses a box filter needs to evaluate the area of the intersection of a given pixel and the surfaces visible in it. The visible intersections can be generated by an appropriate object precision visibility calculation technique (Catmull used the Weiler–Atherton method in his anti-aliasing algorithm [Cat78]) if the window is set such that it covers a single pixel. The color of the resulting pixel is then simply the weighted average of all visible polygon fragments. Although this is a very clear approach theoretically, it is computationally enormously expensive.

A more effective method can take advantage of the fact that regions tend to produce aliases along their edges where the color gradient is high. Thus an anti-aliased polygon generation method can be composed of an anti-aliasing line drawing algorithms to produce the edges, and a normal polygon filling method to draw all the internal pixels. Note that edge drawing must precede interior filling, since only the outer side of the edges should be filtered.

11.2.2 Pre-filtering lines

Recall that non-anti-aliased line segments with a slant of between 0 and 45 degrees are drawn by setting the color of those pixels in each column which are the closest to the line. This method can also be regarded as the point sampling of a one-pixel wide line segment.

Anti-aliasing line drawing algorithms, on the other hand, have to calculate an integral over the intersection of the one-pixel wide line and a finite region centered around the pixel concerned depending on the selected filter type.

Box-filtering lines

For box filtering, the intersection of the one-pixel wide line segment and the pixel concerned has to calculated. Looking at figure 11.3, we can see that a maximum of three pixels may intersect a pixel rectangle in each column if the slant is between 0 and 45 degrees. Let the vertical distance of the three closest pixels to the center of the line be r,s and t respectively, and suppose $s < t \leq r$. By geometric considerations $s, t < 1$, $s + t = 1$ and $r \geq 1$ should also hold.
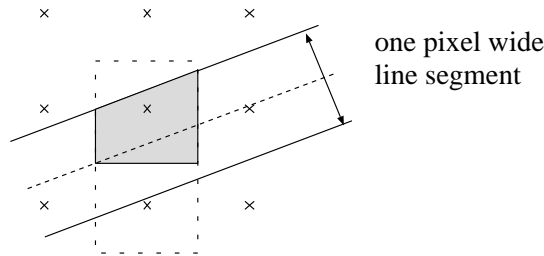
Figure 11.3: Box filtering of a line segment

Unfortunately, the areas of intersection, A_s, A_t and A_r, depend not only on r,s and t, but also on the slant of the line segment. This dependence, however, can be rendered unimportant by using the following approximation:

$$A_s \approx (1 - s), \qquad A_t \approx (1 - t), \qquad A_r \approx 0 \qquad (11.15)$$

These equations are accurate only if the line segment is horizontal, but can be accepted as fair approximations for lines with a slant from 0 to 45 degrees. Variables s and t are calculated for a line $y = m \cdot x + b$:

$$s = m \cdot x + b - \text{Round}(m \cdot x + b) = \text{Error}(x) \quad \text{and} \quad t = 1 - s \quad (11.16)$$

where $\text{Error}(x)$ is, in fact, the accuracy of the digital approximation of the line for vertical coordinate x. The color contribution of the two closest pixels in this pixel column is:

$$I_s = I \cdot (1 - \text{Error}(x)), \qquad I_t = I \cdot \text{Error}(x) \qquad (11.17)$$

(I stands for any color coordinate R, G or B).

These formulae are also primary candidates for incremental evaluation, since if the closest pixel has the same y coordinate for an $x + 1$ as for x:

$$I_s(x + 1) = I_s(x) - I \cdot m, \qquad I_t(x + 1) = I_t(x) + I \cdot m. \qquad (11.18)$$

If the y coordinate has been incremented when stepping from x to $x + 1$, then:

$$I_s(x + 1) = I_s(x) - I \cdot m + I, \qquad I_t(x + 1) = I_t(x) + I \cdot m - I. \qquad (11.19)$$

The color computation can be combined with an incremental y coordinate calculation algorithm, such as the Bresenham's line generator:

AntiAliasedBresenhamLine(x_1, y_1, x_2, y_2, I)
    $\Delta x = x_2 - x_1; \Delta y = y_2 - y_1;$
    $E = -2\Delta x;$
    $dE^+ = 2(\Delta y - \Delta x); dE^- = 2\Delta y;$
    $dI^- = \Delta y / \Delta x \cdot I; dI^+ = I - dI^-;$
    $I_s = I + dI^-; I_t = -dI^-;$
    $y = y_1;$
    for $x = x_1$ **to** x_2 **do**
        if $E \leq 0$ **then** $E \mathrel{+}= dE^-; I_s \mathrel{-}= dI^-; I_t \mathrel{+}= dI^-;$
        else $E \mathrel{+}= dE^+; I_s \mathrel{+}= dI^+; I_t \mathrel{-}= dI^+; y{+}{+};$
        Add Frame Buffer$(x, y, I_s);$
        Add Frame Buffer$(x, y + 1, I_t);$
    endfor

This algorithm assumes that the frame buffer is initialized such that each pixel has the color derived without taking this new line into account, and thus the new contribution can simply be added to it. This is true only if the frame buffer is initialized to the color of a black background and lines do not cross each other. The artifact resulting from crossed lines is usually negligible.

In general cases I must rather be regarded as a weight value determining the portions of the new line color and the color already stored in the frame buffer, which corresponds to the color of objects behind the new line.

The program line "Add Frame Buffer(x, y, I)" should be replaced by the following:

> $\text{color}_{\text{old}} = \text{frame\_buffer}[x, y]$;
> $\text{frame\_buffer}[x, y] = \text{color}_{\text{line}} \cdot I + \text{color}_{\text{old}} \cdot (1 - I)$;

These statements must be executed for each color coordinate R, G, B.

Cone filtering lines

For cone filtering, the volume of the intersection between the one-pixel wide line segment and the one pixel radius cone centered around the pixel concerned has to be calculated. The height of the cone must be selected to guarantee that the volume of the cone is 1. Looking at figure 11.4, we can see that a maximum of three pixels may have intersection with a base circle of the cone in each column if the slant is between 0 and 45 degrees.
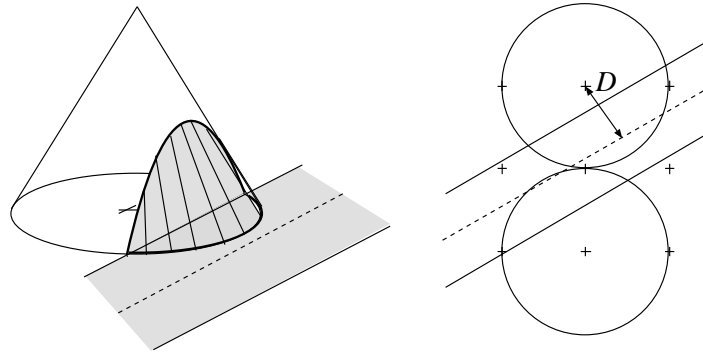
Figure 11.4: Cone filtering of a line segment

Let the distance between the pixel center and the center of the line be D. For possible intersection, D must be in the range of $[-1.5..1.5]$. For a pixel center (X, Y), the convolution integral — that is the volume of the cone segment above a pixel — depends only on the value of D, thus it can be computed for discrete D values and stored in a lookup table $V(D)$ during the design of the algorithm. The number of table entries depends on the number of intensity levels available to render lines, which in turn determines the necessary precision of the representation of D. Since 8–16 intensity levels

are enough to eliminate the aliasing, the lookup table is defined here for three and four fractional bits. Since function $V(D)$ is obviously symmetrical, the number of necessary table entries for three and four fractional bits is $1.5 \cdot 2^3 = 12$ and $1.5 \cdot 2^4 = 24$ respectively. The precomputed $V(D)$ tables, for 3 and 4 fractional bits, are shown in figure 11.5.

D_3	0	1	2	3	4	5	6	7	8	9	10	11
$V(D_3)$	7	6	6	5	4	3	2	1	1	0	0	0

D_4	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$V(D_4)$	14	14	13	13	12	12	11	10	9	8	7	6	5	4	3	3	2	2	1	1	0	0	0	0

Figure 11.5: Precomputed $V(D)$ weight tables

Now the business of the generation of D and the subsequent pixel coordinates must be discussed. Gupta and Sproull [GSS81] proposed the Bresenham algorithm to produce the pixel addresses and introduced an incremental scheme to generate the subsequent D distances.
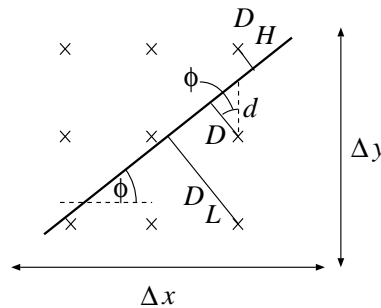
Figure 11.6: Incremental calculation of distance D

Let the obliqueness of the line be ϕ, and the vertical distance between the center of the line and the closest pixel be d (note that for the sake of

simplicity only lines with obliquities in the range of [0..45] are considered as in the previous sections).

For geometric reasons, illustrated by figure 11.6, the D values for the three vertically arranged pixels are:

$$D \;=\; d \cdot \cos \phi = \frac{d \cdot \Delta x}{\sqrt{(\Delta x)^2 + (\Delta y)^2}},$$

$$D_H \;=\; (1-d) \cdot \cos \phi = -D + \frac{\Delta x}{\sqrt{(\Delta x)^2 + (\Delta y)^2}},$$

$$D_L \;=\; (1+d) \cdot \cos \phi = D + \frac{\Delta x}{\sqrt{(\Delta x)^2 + (\Delta y)^2}}. \tag{11.20}$$

A direct correspondence can be established between the distance variable d and the integer error variable E of the Bresenham line drawing algorithm that generates the y coordinates of the subsequent pixels (see section 2.3). The k fractional error variable of the Bresenham algorithm is the required distance plus 0.5 to replace the rounding operation by a simpler truncation, thus $d = k - 0.5$. If overflow happens in k, then $d = k - 1.5$. Using the definition of the integer error variable E, and supposing that there is no overflow in k, the correspondence between E and d is:

$$E = 2\Delta x \cdot (k-1) = 2\Delta x \cdot (d - 0.5) \quad \Longrightarrow \quad 2d \cdot \Delta x = E + \Delta x. \tag{11.21}$$

If overflow happens in k, then:

$$E = 2\Delta x \cdot (k-1) = 2\Delta x \cdot (d + 0.5) \quad \Longrightarrow \quad 2d \cdot \Delta x = E - \Delta x. \tag{11.22}$$

These formulae allow the incremental calculation of $2d \cdot \Delta x$; thus in equation 11.20 the numerators and denominators must be multiplied by two.

The complicated operations including divisions and a square root should be executed once for the whole line, thus the pixel level algorithms contain just simple instructions and a single multiplication not counting the averaging with the colors already stored in the frame buffer. In the subsequent program, expressions that are difficult to calculate are evaluated at the beginning, and stored in the following variables:

$$\text{denom} = \frac{1}{2\sqrt{(\Delta x)^2 + (\Delta y)^2}}, \qquad \Delta D = \frac{2\Delta x}{2\sqrt{(\Delta x)^2 + (\Delta y)^2}}. \tag{11.23}$$

In each cycle nume $= 2d \cdot \Delta x$ is determined by the incremental formulae.

The complete algorithm is:

GuptaSproullLine(x_1, y_1, x_2, y_2, I)
    $\Delta x = x_2 - x_1;\ \Delta y = y_2 - y_1;$
    $E = -\Delta x;$
    $dE^+ = 2(\Delta y - \Delta x);\ dE^- = 2\Delta y;$
    $\text{denom} = 1/(2\sqrt{(\Delta x)^2 + (\Delta y)^2});$
    $\Delta D = 2 \cdot \Delta x \cdot \text{denom};$
    $y = y_1;$
    for $x = x_1$ **to** x_2 **do**
        if $E \le 0$ **then** $\text{nume} = E + \Delta x;\ E\ += dE^-;$
        else $\text{nume} = E - \Delta x;\ E\ += dE^+;\ y{+}{+};$
        $D = \text{nume} \cdot \text{denom};$
        $D_L = D + \Delta D;\ D_H = -D + \Delta D;$
        Add Frame Buffer$(x, y, V(D));$
        Add Frame Buffer$(x, y + 1, V(D_H));$
        Add Frame Buffer$(x, y - 1, V(D_L));$
    endfor
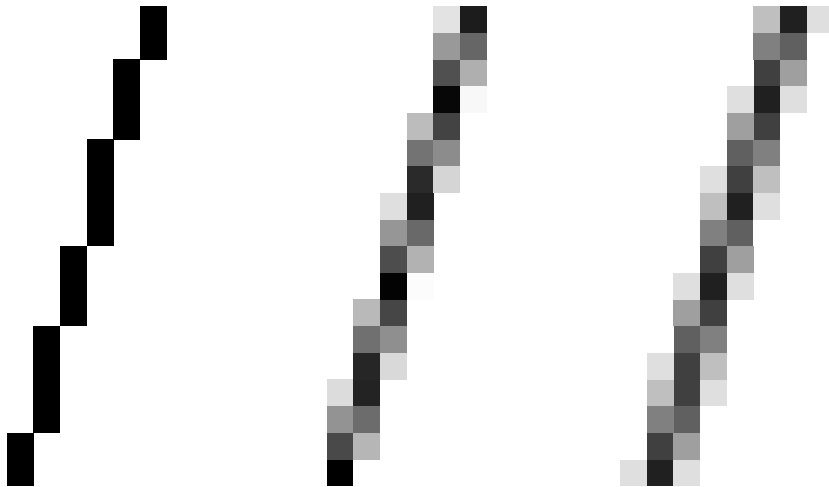
Figure 11.7: Comparison of normal, box-filtered and cone-filtered lines

11.3   Post-filtering anti-aliasing techniques

Post-filtering methods sample the image at a higher sample rate than needed by the resolution of the display hardware ($\Delta x = 1/N, \Delta y = 1/N$), then some digital filtering algorithm is used to calculate pixel colors.

For digital filtering, the spatial integrals of convolution are replaced by infinite sums:

$$I_{sf}(X,Y) = [I(x,y) \sum_i \sum_j \delta(x - i \cdot \Delta x, y - j \cdot \Delta y)] * f(x,y)|_{x=X,y=Y} =$$

$$\sum_i \sum_j I(i \cdot \Delta x, j \cdot \Delta y) \cdot f(X - i \cdot \Delta x, Y - j \cdot \Delta y). \qquad (11.24)$$

Finite extent digital filters simplify the infinite sums to finite expressions. One of the simplest digital filters is the discrete equivalent of the continuous box filter:

$$I_{s,\text{box}}(X,Y) = \frac{1}{(N+1)^2} \sum_{i=-N/2}^{N/2} \sum_{j=-N/2}^{N/2} I(X - i \cdot \Delta x, Y - j \cdot \Delta y). \quad (11.25)$$

This expression states that the average of **subpixel** colors must be taken to produce the color of the real pixel. The color of the subpixels can be determined by a normal, non-anti-aliasing image generation algorithm. Thus, ordinary image synthesis methods can be used, but at a higher resolution, to produce anti-aliased pictures since the anti-aliasing is provided by the final step reducing the resolution to meet the requirements of the display hardware. One may think that this method has the serious drawback of requiring a great amount of additional memory to store the image at a higher resolution, but that is not necessarily true. Ray tracing, for example, generates the image on a pixel-by-pixel basis. When all the subpixels affecting a pixel have been calculated, the pixel color can be evaluated and written into the raster memory, and the very same extra memory can be used again for the subpixels of other pixels. Scan-line methods, on the other hand, require those subpixels which may contribute to the real pixels in the scan-line to be calculated and stored. For the next scan-line, the same subpixel memory can be used again.

As has been stated, discrete algorithms have linear complexity in terms of the pixel number of the image. From that perspective, supersampling

may increase the computational time by a factor of the number of subpixels affecting a real pixel, which is usually not justified by the improvement of image quality, because aliasing is concentrated mainly around sharp edges, and the filtering does not make any significant difference in great homogeneous areas. Therefore, it is worth examining whether the color gradient is great in the neighborhood of a pixel, or whether instead the color is nearly constant, and thus dividing the pixels into subpixels only if required by a high color gradient. This method is called **adaptive supersampling**.

Finally, it is worth mentioning that jaggies can be greatly reduced without increasing the sample frequency at all, simply by moving the sample points from the middle of pixels to the corner of pixels, and generating the color of the pixel as the average of the colors of its corners. Since pixels have four corners, and each internal corner point belongs to four pixels, the number of corner points is only slightly greater than the number of pixel centers. Although this method is not superior in eliminating aliases, it does have a better reconstruction filter for reducing the sharp edges of jaggies.

11.4   Stochastic sampling

Sampling methods applying regular grids produce regularly spaced artifacts that are easily detected by the human eye, since it is especially sensitive to regular and periodic signals. Random placement of sample locations can break up the periodicity of the aliasing artifacts, converting the aliasing effects to random noise which is more tolerable for human observers. Two types of random sampling patterns have been proposed [Coo86], namely the **Poisson disk distribution** and the **jittered** sampling.

11.4.1   Poisson disk distribution

Poisson disk distribution is, in fact, the simulation of the sampling process of the human eye [Yel83]. In effect, it places the sample points randomly with the restriction that the distances of the samples are greater than a specified minimum. Poisson disk distribution has a characteristic distribution in the frequency domain consisting of a spike at zero frequency and a uniform noise beyond the Nyquist limit. Signals having white-noise-like spectrum at higher frequencies, but low-frequency attenuation, are usually

regarded as **blue noise**. This low-frequency attenuation is responsible for the approximation of the minimal distance constraint of Poisson disk distribution.

The sampling process by a Poisson disk distributed grid can be understood as follows: Sampling is, in fact, a multiplication by a "comb function" of the sampling grid. In the frequency domain it is equivalent to convolution by the Fourier transform of this "comb function", which is a blue-noise-like spectrum for Poisson disk distribution except for the spike at 0. Signal components below the Nyquist limit are not affected by this convolution, but components above this limit are turned to a wide range spectrum of noise. Signal components not meeting the requirements of the sampling theorem have been traded off for noise, and thus aliasing in the form of periodic signals can be avoided.

Sampling by Poisson disk distribution is very expensive computationally. One way of approximating appropriate sampling points is based on error diffusion dithering algorithms (see section 11.5 on reduction of quantization effects), since dithering is somehow similar to this process [Mit87], but now the sampling position must be dithered.

11.4.2 Jittered sampling

Jittered sampling is based on a regular sampling grid which is perturbed slightly by random noise [Bal62]. Unlike the application of dithering algorithms, the perturbations are now assumed to be independent random variables. Compared to Poisson disk sampling its result is admittedly not quite as good, but it is less expensive computationally and is well suited to image generation algorithms designed for regular sampling grids.

For notational simplicity, the theory of jittered sampling will be discussed in one-dimension. Suppose function $g(t)$ is sampled and then reconstructed by an ideal low-pass filter. The perturbations of the various sample locations are assumed to be uncorrelated random variables defined by the probability density function $p(x)$. The effect of jittering can be simulated by replacing $g(t)$ by $g(t - \xi(t))$, and sampling it by a regular grid, where function $\xi(t)$ is an independent stochastic process whose probability density function, for any t, is $p(x)$ (figure 11.8).

Jittered sampling can be analyzed by comparing the spectral power distributions of $g(t - \xi(t))$ and $g(t)$.
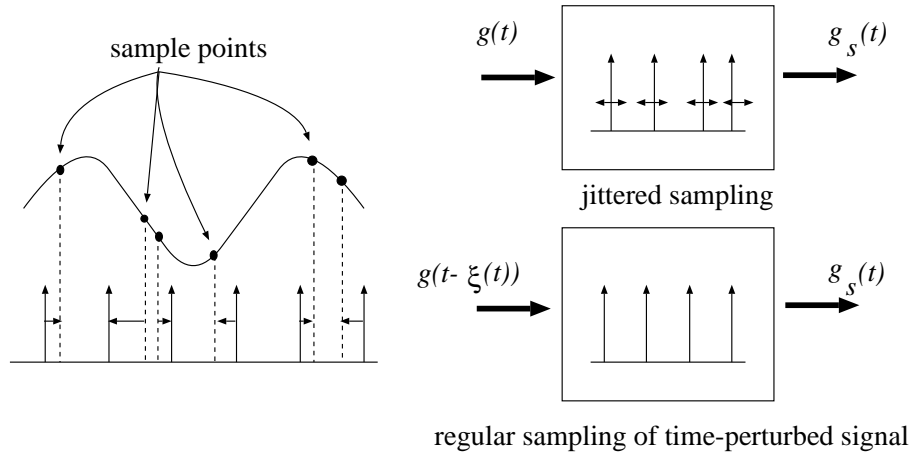
Figure 11.8: Signal processing model of jittered sampling

Since $g(t - \xi(t))$ is a random process, if it were stationary and ergodic [Lam72], then its frequency distribution would be best described by the power density spectrum which is the Fourier transform of its autocorrelation function.

The autocorrelation function of $g(t - \xi(t))$ is derived as an expectation value for any $\tau \neq 0$, taking into account that $\xi(t)$ and $\xi(t + \tau)$ are stochastically independent random variables:

$$R(t, \tau) = E[g(t - \xi(t)) \cdot g(t + \tau - \xi(t + \tau))] =$$

$$\int\limits_{x} \int\limits_{y} g(t - x) \cdot g(t + \tau - y) \cdot p(x) \cdot p(y) \; dx \; dy = (g * p)|_t \cdot (g * p)|_{t+\tau} \quad (11.26)$$

where $g * p$ is the convolution of the two functions. If $\tau = 0$, then:

$$R(t, 0) = E[g(t - \xi(t))^2] = \int\limits_{x} g^2(t - x) \cdot p(x) \; dx = (g^2 * p)|_t. \quad (11.27)$$

Thus the autocorrelation function of $g(t - \xi(t))$ for any τ is:

$$R(t, \tau) = (g * p)|_t \cdot (g * p)|_{t+\tau} + [(g^2 * p) - (g * p)^2]|_t \cdot \delta(\tau) \quad (11.28)$$

where $\delta(\tau)$ is the delta function which is 1 for $\tau = 0$ and 0 for $\tau \neq 0$. This delta function introduces an "impulse" in the autocorrelation function at $\tau = 0$.

Assuming $t = 0$ the size of the impulse at $\tau = 0$ can be given an interesting interpretation if $p(x)$ is an even function ($p(x) = p(-x)$).

$$[(g^2 * p) - (g * p)^2]|_{t=0} = \int_x g^2(-x) \cdot p(x)dx - [\int_x g(-x) \cdot p(x)dx]^2 =$$

$$E[g^2(\xi)] - E^2[g(\xi)] = \sigma^2_{g(\xi)}. \tag{11.29}$$

Hence, the size of the impulse in the autocorrelation function is the variance of the random variable $g(\xi)$. Moving the origin of the coordinate system to t we can conclude that the size of the impulse is generally the variance of the random variable $g(t - \xi(t))$.

Unfortunately $g(t - \xi(t))$ is usually not a stationary process, thus in order to analyze its spectral properties, the power density spectrum is calculated from the "average" autocorrelation function which is defined as:

$$\hat{R}(\tau) = \lim_{T \to \infty} \frac{1}{2T} \int_{-T}^{T} R(t, \tau) \, dt. \tag{11.30}$$

The "average" power density of $g(t - \xi(t))$, supposing $p(x)$ to be even, can be expressed using the definition of the Fourier transform and some identity relations:

$$\hat{S}(f) = \mathcal{F}\hat{R}(\tau) = \lim_{T \to \infty} \frac{1}{2T} [\mathcal{F}_T \ (g * p)]^* \cdot [\mathcal{F}(g * p)] + \overline{\sigma^2_{g(\xi)}} \tag{11.31}$$

where superscript $*$ means the conjugate complex pair of a number, $\overline{\sigma^2_{g(\xi)}}$ is the average variance of the random variable $g(t - \xi(t))$ for different t values, and $\mathcal{F}_T$ stands for the limited Fourier transform defined by the following equation:

$$\mathcal{F}_T \ x(t) = \int_{-T}^{T} x(t) \cdot e^{-2\pi j f t} \, dt, \qquad j = \sqrt{-1} \tag{11.32}$$

Let us compare this power density ($\hat{S}(f)$) of the time perturbed signal with the power density of the original function $g(t)$, which can be defined as follows:

$$S_g(f) = \lim_{T \to \infty} \frac{1}{2T} |\mathcal{F}_T \ g(t)|^2. \tag{11.33}$$

This can be substituted into equation 11.31 yielding:

$$\hat{S}(f) = \frac{|\mathcal{F}(g*p)|^2}{|\mathcal{F}g|^2} \cdot S_g(f) + \overline{\sigma^2_{g(\xi)}}. \tag{11.34}$$

The spectrum consists of a part proportional to the spectrum of the unperturbed $g(t)$ signal and an additive noise carrying $\overline{\sigma^2_{g(\xi)}}$ power in a unit frequency range. Thus the perturbation of time can, in fact, be modeled by a linear network or filter and some additive noise (Figure 11.9).

The gain of the filter perturbing the time variable by an independent random process can be calculated as the ratio of the power density distributions of $g(t)$ and $g(t - \xi(t))$ ignoring the additive noise:

$$\mathrm{Gain}(f) = \frac{|\mathcal{F}(g*p)|^2}{|\mathcal{F}g|^2} = \frac{|\mathcal{F}g|^2 \cdot |\mathcal{F}p|^2}{|\mathcal{F}g|^2} = |\mathcal{F}p|^2. \tag{11.35}$$

Thus, the gain is the Fourier transform of the probability density used for jittering the time.
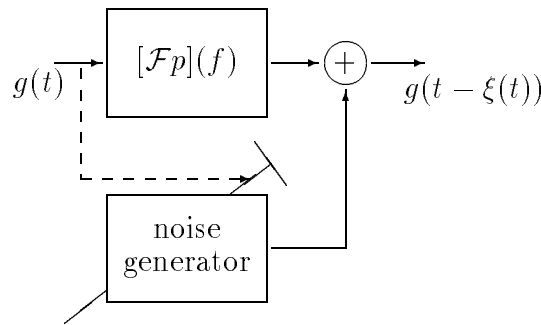
Figure 11.9: System model of time perturbation

Two types of jitters are often used in practice:

1. **White noise jitter**, which distributes the values uniformly between $-T/2$ and $T/2$, where T is the periodicity of the regular sampling grid. The gain of the white noise jitter is:

$$\mathrm{Gain_{wn}}(f) = [\frac{\sin \pi f T}{\pi f T}]^2. \tag{11.36}$$

2. **Gaussian jitter**, which selects the sample points by Gaussian distribution with variance ρ^2.

$$\mathrm{Gain}_{\mathrm{gauss}}(f) = e^{-(2\pi f \rho)^2}. \tag{11.37}$$

Both the white noise jitter and the Gaussian jitter (if $\rho \approx T/6$) are fairly good low-pass filters suppressing the spectrum of the sampled signal above the Nyquist limit, and thus greatly reducing aliasing artifacts.

Jittering trades off aliasing for noise. In order to intuitively explain this result, let us consider the time perturbation for a sine wave. If the extent of the possible perturbations is less than the length of half a period of the sine wave, the perturbation does not change the basic shape of the signal, just distorts it a little bit. The level of distortion depends on the extent of the perturbation and the "average derivative" of the perturbed function as suggested by the formula of the noise intensity defining it as the variance $\overline{\sigma^2_{g(\xi)}}$. If the extent of the perturbations exceeds the length of period, the result is an almost random value in place of the amplitude. The sine wave has disappeared from the signal, only the noise remains.

11.5 Reduction of quantization effects

In digital data processing, not only the number of data must be finite, but also the information represented by a single data element. Thus in computer graphics we have to deal with problems posed by the fact that color information can be represented by a few discrete levels in addition to the finite sampling which allows for the calculation of this color at discrete points only.

In figure 11.10 the color distribution of a shaded sphere is shown. The ideal continuous color is sampled and quantized according to the pixel resolution and the number of quantization levels resulting in a stair-like function in the color space. (Note that aliasing caused stair-like jaggies in pixel space.) The width of these stair-steps is usually equal to the size of many pixels if there are not too many quantization levels, which makes the effect clearly noticeable in the form of quasi-concentric circles on the surface of our sphere. Cheaper graphics systems use eight bits for the representation of a single pixel allowing R, G, B color coordinates to be described by
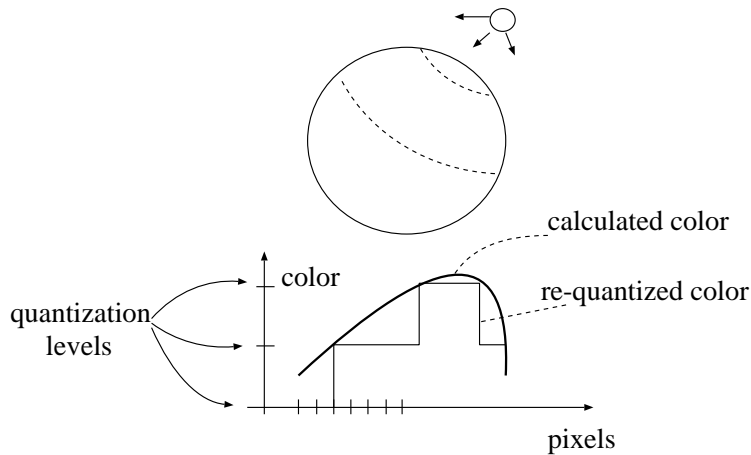
Figure 11.10: Quantization effects

three, three and two bits respectively in true color mode, which is far from adequate. Expensive workstations provide eight bits for every single color coordinate, that is 24 bits for a pixel, making it possible to produce over sixteen million colors simultaneously on the computer screen, but this is still less than the number of colors that can be distinguished by the human eye.

If we have just a limited set of colors but want to produce more, the obvious solution is to try to mix new ones from the available set. At first we might think that this mixing is beyond the capabilities of computer graphics, because the available set of colors is on the computer screen, and thus the mixing should happen when the eye perceives these colors, something which seemingly cannot be controlled from inside the computer. This is fortunately not exactly true. Mixing means a weighted average which can be realized by a low-pass filter, and the eye is known to be a fairly good low-pass filter. Thus, if the color information is provided in such a way that high frequency variation of color is generated where mixing is required, the eye will filter these variations and "compute" its average which exactly amounts to a mixed color.

These high-frequency variations can be produced by either sacrificing the resolution or without decreasing it at all. The respective methods are called **halftoning** and **dithering**.

11.5.1 Halftoning

Halftoning is a well-known technique in the printing industry where gray-level images are produced by placing black points onto the paper, keeping the density of these points proportional to the desired gray level. On the computer screen the same effect can be simulated if adjacent pixels are grouped together to form *logical pixels*. The color of a logical pixel is generated by the pattern of the colors of its physical pixels. Using an $n \times n$ array of bi-level physical pixels, the number of producible colors is $n^2 + 1$ for the price of reducing the resolution by a factor of n in both directions (figure 11.11). This idea can be applied to interpolate between any two subsequent quantization levels (even for any two colors, but this is not used in practice).
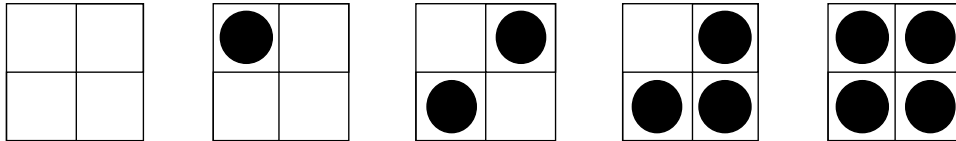
Figure 11.11: Halftone patterns for $n = 4$

11.5.2 Dithering

Unlike halftoning, dithering does not reduce the effective resolution of the display. This technique was originated in measuring theory where the goal was the improvement of the effective resolution of A/D converters. Suppose we have a one-bit quantization unit (a comparator), and a slowly changing signal needs to be measured. Is it possible to say more about the signal than to determine whether it is above or below the threshold level of the comparator?

In fact, the value of the slowly changing signal can be measured accurately if another symmetrical signal, called a dither, having a mean of 0 and appropriate peak value, is added to the signal before quantization. The perturbed signal will spend some of the time below, while the rest remains above the threshold level of the comparator (figure 11.12). The respective times — that is the average or the filtered composite signal — will show
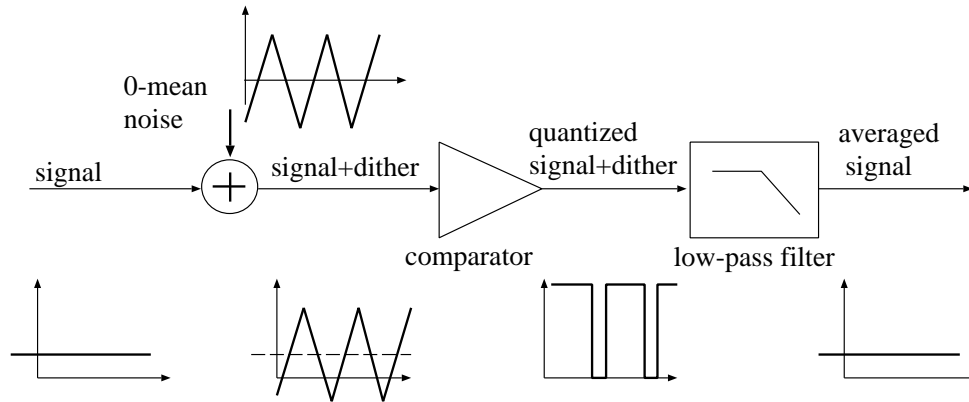
Figure 11.12: Measuring the mean value of a signal by a one-bit quantizer

the mean value of the original signal accurately if the filtering process eliminates the higher frequencies of the dither signal but does not interfere with the low frequency range of the original signal. Thus the frequency characteristic of the dither must be carefully defined: it should contain only high frequency components; that is, it should be **blue noise**.

This idea can readily be applied in computer graphics as well. Suppose the color coordinates of pixels are calculated at a higher level of accuracy than is needed by the frame buffer storage. Let us assume that the frame buffer represents each R, G, B value by n bits and the color computation results in values of $n + d$ bit precision. This can be regarded as a fixed point representation of the colors with d number of fractional bits. Simple truncation would cut off the low d bits, but before truncation a dither signal is added, which is uniformly distributed in the range of [0..1]; that is, it eventually produces distribution in the range of [-0.5..0.5] if truncation is also taken into consideration. This added signal can either be a random or a deterministic function. Periodic deterministic dither functions are also called **ordered dithers**. Taking into account the blue noise criterion, the dither must be a high frequency signal. The maximal frequency dithers are those which have different values on adjacent pixels, and are preferably not periodic. In this context, ordered dithers are not optimal, but they allow for simple hardware implementation, and thus they are the most frequently used methods of reducing the quantization effects in computer graphics.

The averaging of the dithered colors to produce mixed colors is left to the human eye as in halftoning.

Ordered dithers

The behavior of ordered dithers is defined by a periodic function $D(i,j)$ which must be added to the color computed at higher precision. Let the periodicity of this function be N in both vertical and horizontal directions, so D can thus conveniently be described by an $N \times N$ matrix called a dither table. The dithering operation for any color coordinate I is then:

$$I[X,Y] \Leftarrow \text{Trunc}(I[X,Y] + D[X \bmod N, Y \bmod N]). \qquad (11.38)$$

The expectations of a "good" dither can be summarized as follows:

1. It should approximate blue noise, that is, neighboring values should not be too close.

2. It should prevent periodic effects.

3. It must contain uniformly distributed values in the range of $[0..1]$. If fixed point representation is used with d fractional bits, the decimal equivalents of the codes must be uniformly distributed in the range of $[0..2^d]$.

4. The computation of the dithered color must be simple, fast and appropriate for hardware realization. This requirement has two consequences. First, the precision of the fractional representation should correspond to the number of elements in the dither matrix to avoid superfluous bits in the representation. Secondly, dithering requires two modulo N divisions which are easy to accomplish if N is a power of two.

A dither which meets the above requirements would be:

$$D^{(4)} = \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix} \qquad (11.39)$$

where a four-bit fractional representation was assumed, that is, to calculate the real value equivalents of the dither, the matrix elements must be divided by 16.

Let us denote the low k bits and the high k bits of a binary number B by $B|_k$ and $B|^k$ respectively. The complete dithering algorithm is then:

Calculate the (R, G, B) color of pixel (X, Y) and
represent it in an n-integer-bit + 4-fractional-bit form;
$R = (R + D[X|_2, Y|_2])|^n$;
$G = (G + D[X|_2, Y|_2])|^n$;
$B = (B + D[X|_2, Y|_2])|^n$;
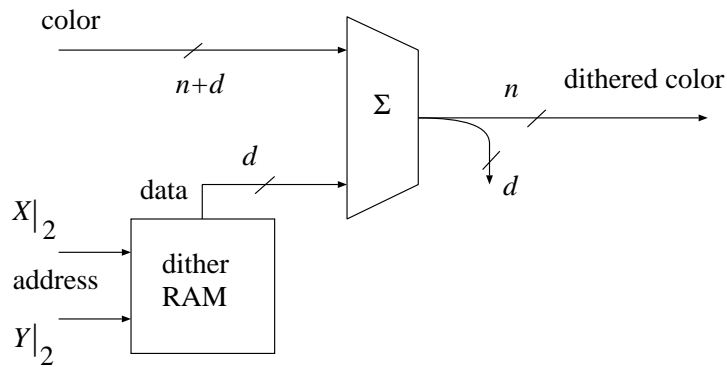
Figure 11.13: Dithering hardware

This expression can readily be implemented in hardware as is shown in figure 11.13.

Chapter 12

TEXTURE MAPPING

The shading equation contains several parameters referring to the optical properties of the surface interacting with the light, including k_a ambient, k_d diffuse, k_s specular, k_r reflective, k_t transmissive coefficients, ν index of refraction etc. These parameters are not necessarily constant over the surface, thus allowing surface details, called **textures**, to appear on computer generated images. Texture mapping requires the determination of the surface parameters each time the shading equation is calculated for a point on the surface. Recall that for ray tracing, the ray-object intersection calculation provides the visible surface point in the world coordinate system. For incremental methods, however, the surfaces are transformed to the screen coordinate system where the visibility problem is solved, and thus the surface points to be shaded are defined in screen coordinates.

Recall, too, that the shading equation is evaluated in the world coordinate system even if the visible surface points are generated in screen space, because the world-screen transformation usually modifies the angle vectors needed for the shading equation. For directional and ambient lightsource only models, the shading equation can also be evaluated in screen space, but the normal vectors defined in the world coordinate system must be used.

The varying optical parameters required by the shading equation, on the other hand, are usually defined and stored in a separate coordinate system, called **texture space**. The texture information can be represented by some data stored in an array or by a function that returns the value needed for the points of the texture space. In order for there to be a correspondence between texture space data and the points of the surface, a transformation

is associated with the texture, which maps texture space onto the surface defined in its local coordinate system. This transformation is called **parameterization**.

Modeling transformation maps this local coordinate system point to the world coordinate system where the shading is calculated. In ray tracing the visibility problem is also solved here. Incremental shading models, however, need another transformation from world coordinates to screen space where the hidden surface elimination and simplified color computation take place. This latter mapping is regarded as **projection** in texture mapping (figure 12.1).
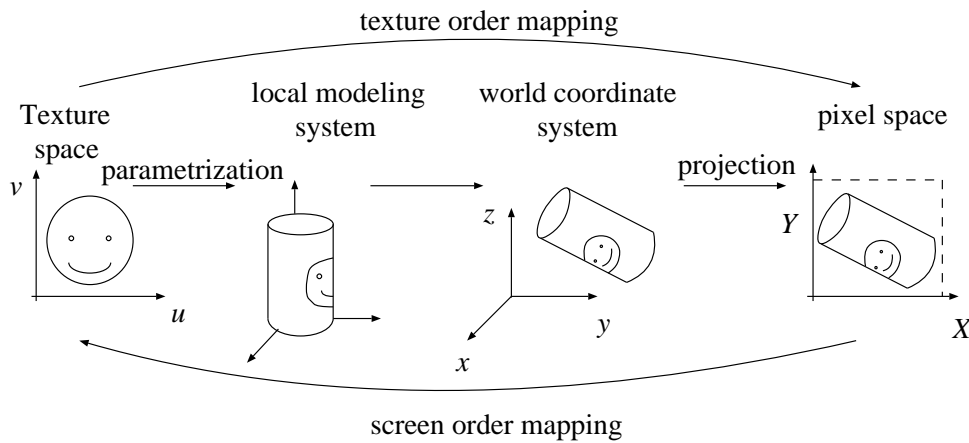
Figure 12.1: Survey of texture mapping

Since the parameters of the shading equation are required in screen space, but available only in texture space, the mapping between the two spaces must be evaluated for each pixel to be shaded.

Generally two major implementations are possible:

1. **Texture order** or **direct mapping** which scans the data in texture space and maps from texture space to screen space.

2. **Screen order** or **inverse mapping** which scans the pixels in screen space and uses the mapping from screen space to texture space.

Texture order mapping seems more effective, especially for large textures stored in files, since they access the texture sequentially. Unfortunately, there is no guarantee that, having transformed uniformly sampled data from texture space to screen space, all pixels belonging to a given surface will be produced. Holes and overlaps may occur on the image. The correct sampling of texture space, which produces all pixels needed, is a difficult problem if the transformation is not linear. Since texture order methods access texture elements one after the other they also process surfaces sequentially, making themselves similar to and appropriate for object precision hidden surface algorithms.

Image precision algorithms, on the other hand, evaluate pixels sequentially and thus require screen order techniques and random access of texture maps. A screen order mapping can be very slow if texture elements are stored on a sequential medium, and it needs the calculation of the inverse parameterization which can be rather difficult. Nevertheless, screen order is more popular, because it is appropriate for image precision hidden surface algorithms.

Interestingly, although the z-buffer method is an image precision technique, it is also suitable for texture order mapping, because it processes polygons sequentially.

The texture space can be either one-dimensional, two-dimensional or three-dimensional. A one-dimensional texture has been proposed, for example, to simulate the thin film interference produced on a soap bubble, oil and water [Wat89].

Two-dimensional textures can be generated from frame-grabbed or computer synthesized images and are glued or "wallpapered" onto a three-dimensional object surface. The "wallpapers" will certainly have to be distorted to meet topological requirements. The 2D texture space can generally be regarded as a unit square in the center of a u, v texture coordinate system. Two-dimensional texturing reflects our subjective concept of surface painting, and that is one of the main reasons why it is the most popular texturing method.

Three-dimensional textures, also called **solid textures**, neatly circumvent the parameterization problem, since they define the texture data in the 3D local coordinate system — that is in the same space where the geometry is defined — simplifying the parameterization into an identity transformation. The memory requirements of this approach may be prohibitive, how-

ever, and thus three-dimensional textures are commonly limited to functionally defined types only. Solid texturing is basically the equivalent of carving the object out of a block of material. It places the texture onto the object coherently, not producing discontinuities of texture where two faces meet, as 2D texturing does. The simulation of wood grain on a cube, for example, is only possible by solid texturing in order to avoid discontinuities of the grain along the edges of the cube.

12.1 Parameterization for two-dimensional textures

Parameterization connects the unit square of 2D texture space to the 3D object surface defined in the local modeling coordinate system.

12.1.1 Parameterization of parametric surfaces

The derivation of this transformation is straightforward if the surface is defined parametrically over the unit square by a positional vector function:

$$\vec{r}(u,v) = \left[\begin{array}{c} x(u,v) \\ y(u,v) \\ z(u,v) \end{array} \right]. \tag{12.1}$$

Bezier and bicubic parametric patches fall into this category. For other parametric surfaces, such as B-spline surfaces, or in cases where only a portion of a Bezier surface is worked with, the definition is similar, but texture coordinates come from a rectangle instead of a unit square. These surfaces can also be easily parameterized, since only a linear mapping which transforms the rectangle onto the unit square before applying the parametric functions is required.

For texture order mapping, these formulae can readily be applied in order to obtain corresponding $\vec{r}(u,v)$ 3D points for u,v texture coordinates. For scan order mapping, however, the inverse of $\vec{r}(u,v)$ has to be determined, which requires the solution of a non-linear equation.

12.1.2 Parameterization of implicit surfaces

Parameterization of an implicitly defined surface means the derivation of an explicit equation for that surface. The ranges of the natural parameters may not fall into a unit square, thus making an additional linear mapping necessary. To explain this idea, the examples of the sphere and cylinder are taken.

Parameterization of a sphere

The implicit definition of a sphere around a point (x_c, y_c, z_c) with radius r is:

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2. \tag{12.2}$$

An appropriate parameterization can be derived using a spherical coordinate system with spherical coordinates ϕ and θ.

$$
\begin{aligned}
x(\phi, \theta) &= x_c + r \cdot \cos \theta \cdot \cos \phi, \\
y(\phi, \theta) &= y_c + r \cdot \cos \theta \cdot \sin \phi, \\
z(\phi, \theta) &= z_c + r \cdot \sin \theta.
\end{aligned}
\tag{12.3}
$$

The spherical coordinate ϕ covers the range $[0..2\pi]$, and θ covers the range $[-\pi/2..\pi/2]$, thus, the appropriate (u, v) texture coordinates are derived as follows:

$$u = \frac{\phi}{2\pi}, \qquad v = \frac{(\theta + \pi/2)}{\pi}. \tag{12.4}$$

The complete transformation from texture space to modeling space is:

$$
\begin{aligned}
x(u, v) &= x_c + r \cdot \cos \pi (v - 0.5) \cdot \cos 2\pi u, \\
y(u, v) &= y_c + r \cdot \cos \pi (v - 0.5) \cdot \sin 2\pi u, \\
z(u, v) &= z_c + r \cdot \sin \pi (v - 0.5).
\end{aligned}
\tag{12.5}
$$

For texture order mapping, the inverse transformation is:

$$u(x, y, z) = \frac{1}{2\pi} \cdot \arctan^*(y - y_c, x - x_c),$$

$$v(x, y, z) = \frac{1}{\pi} \cdot (\arcsin \frac{z - z_c}{r} + \pi/2), \tag{12.6}$$

where $\arctan^*(a, b)$ is the extended arctan function, that is, it produces an angle ξ in $[0..2\pi]$ if $\sin \xi = a$ and $\cos \xi = b$.

Parameterization of a cylinder

A cylinder of height H located around the z axis has the following implicit equation:

$$X^2 + Y^2 = r^2, \qquad 0 \le z \le H. \tag{12.7}$$

The same cylinder can be conveniently expressed by cylindrical coordinates ($\theta \in [0..2\pi], h \in [0..H]$):

$$\begin{aligned} X(\theta, h) &= r \cdot \cos \theta, \\ Y(\theta, h) &= r \cdot \sin \theta, \\ Z(\theta, h) &= h. \end{aligned} \tag{12.8}$$

To produce an arbitrary cylinder, this is rotated and translated by an appropriate affine transformation:

$$[x(\theta, h), y(\theta, h), z(\theta, h), 1] = [X(\theta, h), Y(\theta, h), Z(\theta, h), 1] \cdot \begin{bmatrix} & & 0 \\ \mathbf{A}_{3\times 3} & & 0 \\ & & 0 \\ \mathbf{p^T} & & 1 \end{bmatrix} \tag{12.9}$$

where $\mathbf{A}$ must be an orthonormal matrix; that is, its row vectors must be unit vectors and must form a perpendicular triple. Matrices of this type do not alter the shape of the object and thus preserve cylinders.

Since cylindrical coordinates θ and h expand over the ranges $[0..2\pi]$ and $[0, H]$ respectively, the domain of cylindrical coordinates can thus be easily mapped onto a unit square:

$$u = \frac{\theta}{2\pi}, \qquad v = \frac{h}{H}. \tag{12.10}$$

The complete transformation from texture space to modeling space is:

$$[x(u, v), y(u, v), z(u, v), 1] = [r \cdot \cos 2\pi u, r \cdot \sin 2\pi u, v \cdot H, 1] \cdot \begin{bmatrix} & & 0 \\ \mathbf{A}_{3\times 3} & & 0 \\ & & 0 \\ \mathbf{p^T} & & 1 \end{bmatrix}. \tag{12.11}$$

The inverse transformation is:

$$[\alpha, \beta, h, 1] = [x(u,v), y(u,v), z(u,v), 1] \cdot \begin{bmatrix} & 0 \\ \mathbf{A}_{3\times 3} & 0 \\ & 0 \\ \mathbf{p^T} & 1 \end{bmatrix}^{-1}$$

$$u(x,y,z) = u(\alpha, \beta) = \frac{1}{2\pi} \cdot \arctan^*(\beta, \alpha), \qquad v(x,y,z) = \frac{h}{H} \qquad (12.12)$$

where $\arctan^*(a,b)$ is the extended arctan function as before.

12.1.3   Parameterization of polygons

Image generation algorithms, except in the case of ray tracing, suppose object surfaces to be broken down into polygons. This is why texturing and parameterization of polygons are so essential in computer graphics. The parameterization, as a transformation, must map a 2D polygon given by vertices $v_1(u,v), v_2(u,v), ..., v_n(u,v)$ onto a polygon in the 3D space, defined by vertex points $\vec{V}_1(x,y,z), \vec{V}_2(x,y,z), ..., \vec{V}_n(x,y,z)$. Let this transformation be $\mathcal{P}$. As stated, it must transform the vertices to the given points:

$$\vec{V}_1(x,y,z) = \mathcal{P}v_1(u,v), \ \ \vec{V}_2(x,y,z) = \mathcal{P}v_2(u,v), \ldots, \vec{V}_n(x,y,z) = \mathcal{P}v_n(u,v).$$
$$(12.13)$$

Since each vertex $\vec{V}_i$ is represented by three coordinates, equation 12.13 consists of $3n$ equations. These equations, however, are not totally independent, since the polygon is assumed to be on a plane; that is, the plane equation defined by the first three non-collinear vertices

$$(\vec{r} - \vec{V}_1) \cdot (\vec{V}_3 - \vec{V}_1) \times (\vec{V}_2 - \vec{V}_1) = 0$$

should hold, where $\vec{r}$ is any of the other vertices, $\vec{V}_4, \vec{V}_5, ..., \vec{V}_n$. The number of these equations is $n-3$. Thus, if $\mathcal{P}$ guarantees the preservation of polygons, it should have $3n - (n-3)$ free, independently controllable parameters in order to be able to map a 2D n-sided polygon onto an arbitrary 3D n-sided planar polygon. The number of independently controllable parameters is also called the **degree of freedom**.

Function $\mathcal{P}$ must also preserve lines, planes and polygons to be suitable for parameterization. The most simple transformation which meets this requirement is the linear mapping:

$$x = A_x \cdot u + B_x \cdot v + C_x, \;\; y = A_y \cdot u + B_y \cdot v + C_y, \;\; z = A_z \cdot u + B_z \cdot v + C_z. \;\; (12.14)$$

The degree of freedom (number of parameters) of this linear transformation is 9, requiring $3n - (n - 3) \leq 9$, or equivalently $n \leq 3$ to hold. Thus, only triangles ($n = 3$) can be parameterized by linear transformations.

Triangles

$$[x, y, z] = [u, v, 1] \cdot \begin{bmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{bmatrix} = [u, v, 1] \cdot \mathbf{P}. \quad\quad (12.15)$$

The unknown matrix elements can be derived from the solution of a 9×9 system of linear equations developed by putting the coordinates of $\vec{V}_1$, $\vec{V}_2$ and $\vec{V}_3$ into this equation.

For screen order, the inverse transformation is used:

$$[u, v, 1] = [x, y, z] \cdot \begin{bmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{bmatrix}^{-1} = [x, y, z] \cdot \mathbf{P}^{-1}. \quad\quad (12.16)$$

Quadrilaterals

As has been stated, a transformation mapping a quadrilateral from the 2D texture space to the 3D modeling space is generally non-linear, because the degree of freedom of a 2D to 3D linear transformation is less than is required by the placement of four arbitrary points. When looking for an appropriate non-linear transformation, however, the requirement stating that the transformation must preserve polygons has also to be kept in mind.

As for other cases, where the problem outgrows the capabilities of 3D space, 4D homogeneous representation can be relied on again [Hec86], since the number of free parameters is 12 for a linear 2D to 4D transformation, which is more than the necessary limit of 11 derived by inserting n=4 into formula $3n - (n - 3)$. Thus, a linear transformation can be established between a 2D and a 4D polygon.

From 4D space, however, we can get back to real 3D coordinates by a homogeneous division. Although this division reduces the degree of freedom by 1 — scalar multiples of homogeneous matrices are equivalent — the number of free parameters, 11, is still enough. The overall transformation consisting of a matrix multiplication and a homogeneous division has been proven to preserve polygons if the matrix multiplication maps no part of the quadrilateral onto the ideal points (see section 5.1). In order to avoid the wrap-around problem of projective transformations, convex quadrilaterals should not be mapped on concave ones and vice versa.

Using matrix notation, the parameterization transformation is:

$$[x \cdot h, y \cdot h, z \cdot h, h] = [u, v, 1] \cdot \begin{bmatrix} U_x & U_y & U_z & U_h \\ V_x & V_y & V_z & V_h \\ W_x & W_y & W_z & W_h \end{bmatrix} = [u, v, 1] \cdot \mathbf{P_{3 \times 4}}.$$
(12.17)

We arbitrarily choose $W_h = 1$ to select one matrix from the equivalent set. After homogeneous division, we get:

$$x(u, v) = \frac{U_x \cdot u + V_x \cdot v + W_x}{U_h \cdot u + V_h \cdot v + 1},$$

$$y(u, v) = \frac{U_y \cdot u + V_y \cdot v + W_y}{U_h \cdot u + V_h \cdot v + 1},$$

$$z(u, v) = \frac{U_z \cdot u + V_z \cdot v + W_z}{U_h \cdot u + V_h \cdot v + 1}.$$
(12.18)

The inverse transformation, assuming $D_w = 1$, is:

$$[u \cdot w, v \cdot w, w] = [x, y, z, 1] \cdot \begin{bmatrix} A_u & A_v & A_w \\ B_u & B_v & B_w \\ C_u & C_v & C_w \\ D_u & D_v & D_w \end{bmatrix} = [x, y, z, 1] \cdot \mathbf{Q_{4 \times 3}}, \quad (12.19)$$

$$u(x, y, z) = \frac{A_u \cdot x + B_u \cdot y + C_u \cdot z + D_u}{A_w \cdot x + B_w \cdot y + C_w \cdot z + 1},$$

$$v(x, y, z) = \frac{A_v \cdot x + B_v \cdot y + C_v \cdot z + D_v}{A_w \cdot x + B_w \cdot y + C_w \cdot z + 1}.$$
(12.20)

General polygons

The recommended method of parameterization of general polygons subdivides the polygon into triangles (or less probably into quadrilaterals) and generates a parameterization for the separate triangles by the previously discussed method. This method is pretty straightforward, although it maps line segments onto staggered lines, which may cause noticeable artifacts on the image. This effect can be greatly reduced by decreasing the size of the triangles composing the polygon.

Polygon mesh models

The natural way of reducing the dimensionality of a polygon mesh model from three to two is by unfolding it into a two-dimensional folding plane, having separated some of its adjacent faces along the common edges [SSW86]. If the faces are broken down into triangles and quadrilaterals, the texture space can easily be projected onto the folding space, taking into account which texture points should correspond to the vertices of the 2D unfolded object. The edges that must be separated to allow the unfolding of the polygon mesh model can be determined by topological considerations. The adjacency of the faces of a polyhedron can be defined by a graph where the nodes represent the faces or polygons, and the arcs of the graph represent the adjacency relationship of the two faces.
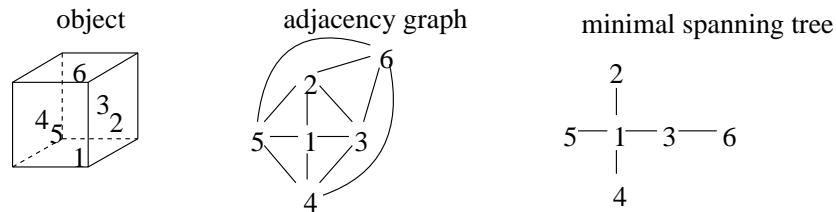
Figure 12.2: Face adjacency graph of a polyhedron

Polygon mesh models whose adjacency graphs are tree-graphs can obviously be unfolded. Thus, in order to prepare for the unfolding operation, those adjacent faces must be separated whose tearing will guarantee that the resulting adjacency graph is a tree. A graph usually has many spanning

trees, thus the preprocessing step can have many solutions. By adding cost information to the various edges, an "optimal" unfolding can be achieved. There are several alternative ways to define the cost values:

- The user specifies which edges are to be preserved as the border of two adjacent polygons. These edges are given 0 unit cost, while the rest are given 1.

- The cost can be defined by the difference between the angle of the adjacent polygons and π. This approach aims to minimize the total rotations in order to keep the 2D unfolded model compact.

There are several straightforward algorithms which are suitable for the generation of a minimal total cost spanning tree of a general graph [Har69]. A possible algorithm builds up the graph incrementally and adds that new edge which has lowest cost from the remaining unused edges and does not cause a cycle to be generated in each step.

The unfolding operation starts at the root of the tree, and the polygons adjacent to it are pivoted about the common edge with the root polygon. When a polygon is rotated around the edge, all polygons adjacent to it must also be rotated (these polygons are the descendants of the given polygon in the tree). Having unfolded all the polygons adjacent to the root polygon, these polygons come to be regarded as roots and the algorithm is repeated for them recursively. The unfolding program is:

UnfoldPolygon(*poly*, *edge*, ϕ);
    Rotate *poly* and all its children around *edge* by $\pi - \phi$;
    for each child of *poly*
        UnfoldPolygon(*child*, edge(*poly, child*), angle(*poly, child*));
    endfor
end
Main program
    for each child of *root*
        UnfoldPolygon(*child*, edge(*root, child*), angle(*root, child*));
    endfor
end

The information regarding the orientation and position of polygons is usually stored in transformation matrices. A new rotation of a polygon means the concatenation of a new transformation matrix to the already developed matrix of the polygon. One of the serious limitations of this approach is that it partly destroys polygonal adjacency; that is, the unfolded surface will have a different topology from the original surface. A common edge of two polygons may be mapped onto two different locations of the texture space, causing discontinuities in texture along polygon boundaries.

This problem can be overcome by the method discussed in the subsequent section.

General surfaces

A general technique developed by Bier and Sloan [BS86] uses an intermediate surface to establish a mapping between the surface and the texture space. When mapping from the texture space to the surface, first the texture point is mapped onto the intermediate surface by its parameterization, then some "natural" projection is used to map the point onto the target surface. The texturing transformation is thus defined by a two-phase mapping.

object normal   object centroid   intermediate surface normal
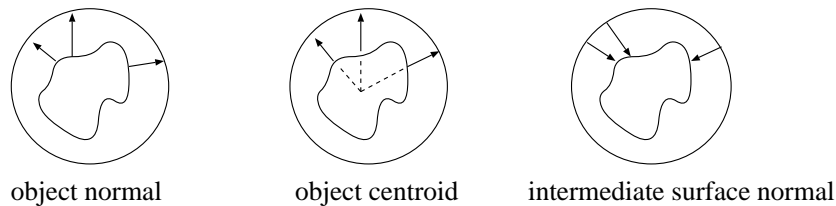
Figure 12.3: Natural projections

The intermediate surface must be easy to parameterize and therefore usually belongs to one of the following categories:

1. Planar polygon

2. Sphere

3. Cylinder

4. Faces of a cube.

The possibilities of the "natural" projection between the intermediate and target surfaces are shown in figure 12.3.

12.2 Texture mapping in ray tracing

Ray tracing determines which surface points must be shaded in the world coordinate system; that is, only the modeling transformation is needed to connect the shading space with the space where the textures are stored. Ray tracing is a typical image precision technique which generates the color of pixels sequentially, thus necessitating a scan order texture mapping method.

Having calculated the object visible along either a primary or secondary ray — that is, having found the nearest intersection of the ray and the objects — the shading equation must be evaluated, which may require the access of texture maps for varying parameters. The derivation of texture coordinates depends not only on the type of texturing, but also on the surface to be rendered.

For parametrically defined patches, such as Bezier and B-spline surfaces, the intersection calculation has already determined the parameters of the surface point to be shaded, thus this information is readily available to fetch the necessary values from 2D texture maps.

For other surface types and for solid texturing, the point should be transformed to local modeling space from the world coordinate system. Solid texture value is generated here usually by calling a function with the local coordinates which returns the required parameters. Two-dimensional textures require the inverse parameterization to be calculated, which can be done by any of the methods so far discussed.

12.3 Texture mapping for incremental shading models

In incremental shading the polygons are supposed to be in the screen coordinate system, and a surface point is thus represented by the (X, Y) pixel coordinates with the depth value Z which is only important for hidden surface elimination, but not necessarily for texture mapping since the definition of the surface, viewing, and the (X, Y) pair of pixel coordinates completely

identify where the point is located on the surface. Incremental methods deal with polygon mesh approximations, never directly with the real equations of explicit or implicit surfaces. Texturing transformations, however, may refer either to the original surfaces or to polygons.

12.3.1   Real surface oriented texturing

If the texture mapping has parameterized the original surfaces rather than their polygon mesh approximations, an applicable scan-order algorithm is very similar to that used for ray tracing. First the point is mapped from screen space to the local modeling coordinate system by the inverse of the composite transformation. Since the composite transformation is homogeneous (or in special cases linear if the projection is orthographic), its inverse is also homogeneous (or linear), as has been proven in section 5.1 on properties of homogeneous transformations. From the modeling space, any of the discussed methods can be used to inversely parameterize the surface and to obtain the corresponding texture space coordinates.

Unlike ray tracing, parametric surfaces may pose serious problems, when the inverse parameterization is calculated, since this requires the solution of a two-variate, non-linear equation $\vec{r}(u,v) = \vec{p}$, where $\vec{p}$ is that point in the modeling space which maps to the actual pixel.

To solve this problem, the extension of the iterative root searching method based on the refinement of the subdivision of the parametric patch into planar polygons can be used here.

Some degree of subdivision has also been necessary for polygon mesh approximation. **Subdivision** of a parametric patch means dividing it along its isoparametric curves. Selecting uniformly distributed u and v parameters, the subdivision yields a mesh of points at the intersection of these curves. Then a mesh of planar polygons (quadrilaterals) can be defined by these points, which will approximate the original surface at a level determined by how exact the isoparametric division was.

When it turns out that a point of a polygon approximating the parametric surface is mapped onto a pixel, a rough approximation of the u,v surface parameters can be derived by looking at which isoparametric lines defined this polygon. This does not necessitate the solution of the non-linear equation if the data of the original subdivision which defines the polygon vertices at the intersection of isoparametric lines are stored somewhere. The inverse

problem for a quadrilateral requires the search for this data only. The search will provide the isoparametric values along the boundaries of the quadrilateral. In order to find the accurate u, v parameters for an inner point, the subdivision must be continued, but without altering or further refining the shape of the approximation of the surface, as proposed by Catmull [Cat74]. In his method, the refinement of the subdivision is a parallel procedure in parameter space and screen space. Each time the quadrilateral in screen space is divided into four similar polygons, the rectangle in the parameter space is also broken down into four rectangles. By a simple comparison it is decided which screen space quadrilateral maps onto the actual pixel, and the algorithm proceeds for the resulted quadrilateral and its corresponding texture space rectangle until the polygon coming from the subdivision covers a single pixel. When the subdivision terminates, the required texture value is obtained from the texture map. Note that it is not an accurate method, since the original surface and the viewing transformation are not linear, but the parallel subdivision used linear interpolation. However, if the original interpolation is not too inexact, then it is usually acceptable.

12.3.2   Polygon based texturing

When discussing parameterization, a correspondence was established between the texture space and the local modeling space. For polygons subdivided into triangles and quadrilaterals, the parameterization and its inverse can be expressed by a homogeneous transformation. The local modeling space, on the other hand, is mapped to the world coordinate system, then to the screen space by modeling and viewing transformations respectively. The concatenation of the modeling and viewing transformations is an affine mapping for orthographic projection and is a homogeneous transformation for perspective projection.

Since both the parameterization and the projection are given by homogeneous transformations, their composition directly connecting texture space with screen space will also be a homogeneous transformation. The matrix representation of this mapping for quadrilaterals and perspective transformation is derived as follows. The parameterization is:

$$[x \cdot h, y \cdot h, z \cdot h, h] = [u, v, 1] \cdot \mathbf{P_{3 \times 4}}. \tag{12.21}$$

The composite modeling and viewing transformation is:

$$[X \cdot q, Y \cdot q, Z \cdot q, q] = [x \cdot h, y \cdot h, z \cdot h, h] \cdot \mathbf{T_{V(4 \times 4)}}. \tag{12.22}$$

Projection will simply ignore the Z coordinate if it is executed in screen space, and thus it is not even worth computing in texture mapping. Since the third column of matrix $\mathbf{T_{V(4 \times 4)}}$ is responsible for generating Z, it can be removed from the matrix:

$$[X \cdot q, Y \cdot q, q] = [x \cdot h, y \cdot h, z \cdot h, h] \cdot \mathbf{T_{V(4 \times 3)}} = [u, v, 1] \cdot \mathbf{P_{3 \times 4}} \cdot \mathbf{T_{V(4 \times 3)}}. \tag{12.23}$$

Denoting $\mathbf{P_{3 \times 4}} \cdot \mathbf{T_{V(4 \times 3)}}$ by $\mathbf{C_{3 \times 3}}$, the composition of parameterization and projection is:

$$[X \cdot q, Y \cdot q, q] = [u, v, 1] \cdot \mathbf{C_{3 \times 3}}. \tag{12.24}$$

The inverse transformation for scan order mapping is:

$$[u \cdot w, v \cdot w, w] = [X, Y, 1] \cdot \mathbf{C_{3 \times 3}^{-1}}. \tag{12.25}$$

Let the element of $\mathbf{C_{3 \times 3}}$ and $\mathbf{C_{3 \times 3}^{-1}}$ in ith row and in jth column be c_{ij} and C_{ij} respectively. Expressing the texture coordinates directly, we can conclude that u and v are quotients of linear expressions of X and Y, while X and Y have similar formulae containing u and v.

The texture order mapping is:

$$X(u, v) = \frac{c_{11} \cdot u + c_{21} \cdot v + c_{31}}{c_{13} \cdot u + c_{23} \cdot v + c_{33}}, \quad Y(u, v) = \frac{c_{12} \cdot u + c_{22} \cdot v + c_{32}}{c_{13} \cdot u + c_{23} \cdot v + c_{33}}. \tag{12.26}$$

The screen order mapping is:

$$u(X, Y) = \frac{C_{11} \cdot X + C_{21} \cdot Y + C_{31}}{C_{13} \cdot X + C_{23} \cdot Y + C_{33}}, \quad v(X, Y) = \frac{C_{12} \cdot X + C_{22} \cdot Y + C_{32}}{C_{13} \cdot X + C_{23} \cdot Y + C_{33}}. \tag{12.27}$$

If triangles are parameterized and orthographic projection is used, both transformations are linear requiring their composite texturing transformation to be linear also. Linearity means that:

$$c_{13}, c_{23} = 0, \quad c_{33} = 1, \quad C_{13}, C_{23} = 0, \quad C_{33} = 1 \tag{12.28}$$

Scan order polygon texturing

When pixel X, Y is shaded in screen space, its corresponding texture coordinates can be derived by evaluating a rational equation 12.27. This can be further optimized by the incremental concept. Let the numerator and the denominator of quotients defining $u(X)$ be $uw(X)$ and $w(X)$ respectively. Although the division cannot be eliminated, $u(X + 1)$ can be calculated from $u(X)$ by two additions and a single division if $uw(X)$ and $w(X)$ are evaluated incrementally:

$$uw(X+1) = uw(X)+C_{11}, \quad w(X+1) = w(X)+C_{13}, \quad u(X+1) = \frac{uw(X + 1)}{w(X + 1)}.$$
$$(12.29)$$

A similar expression holds for the incremental calculation of v. In addition to the incremental calculation of texture coordinates inside the horizontal spans, the incremental concept can also be applied on the starting edge of the screen space triangle. Thus, the main loop of the polygon rendering of Phong shading is made appropriate for incremental texture mapping:

$X_{\text{start}} = X_1 + 0.5; \ X_{\text{end}} = X_1 + 0.5; \ \vec{N}_{\text{start}} = \vec{N}_1;$
$uw_s = uw_1; \ vw_s = vw_1; \ w_s = w_1;$
for $Y = Y_1$ **to** Y_2 **do**
 $uw = uw_s; \ vw = vw_s; \ w = w_s;$
 $\vec{N} = \vec{N}_{\text{start}};$
 for $X = \text{Trunc}(X_{\text{start}})$ **to** $\text{Trunc}(X_{\text{end}})$ **do**
  $u = uw/w; \ v = vw/w;$
  $(R, G, B) = \text{ShadingModel}(\ \vec{N}, \ u, \ v \ );$
  write$(\ X, Y, \text{Trunc}(R), \text{Trunc}(G), \text{Trunc}(B)\ );$
  $\vec{N} \mathrel{+}= \delta\vec{N}_X;$
  $uw \mathrel{+}= C_{11}; \ vw \mathrel{+}= C_{12}; \ w \mathrel{+}= C_{13};$
 endfor
 $X_{\text{start}} \mathrel{+}= \delta X_Y^s; \ X_{\text{end}} \mathrel{+}= \delta X_Y^e; \ \vec{N}_{\text{start}} \mathrel{+}= \delta\vec{N}_Y^s;$
 $uw_s \mathrel{+}= \delta uw_Y^s; \ vw_s \mathrel{+}= \delta vw_Y^s; \ w_s \mathrel{+}= \delta w_Y^s;$
endfor

If the texturing transformation is linear, that is if triangles are parameterized and orthographic projection is used, the denominator is always 1, thus simplifying the incremental formula to a single addition:

$X_{\text{start}} = X_1 + 0.5;\ X_{\text{end}} = X_1 + 0.5;\ \vec{N}_{\text{start}} = \vec{N}_1;$

$u_s = u_1;\ v_s = v_1;$

for $Y = Y_1$ **to** Y_2 **do**

    $u = u_s;\ v = v_s;$

    $\vec{N} = \vec{N}_{\text{start}};$

    for $X = \text{Trunc}(X_{\text{start}})$ **to** $\text{Trunc}(X_{\text{end}})$ **do**

        $(R, G, B) = \text{ShadingModel}(\ \vec{N},\ u,\ v\ );$

        write$(\ X, Y, \text{Trunc}(R), \text{Trunc}(G), \text{Trunc}(B)\ );$

        $\vec{N}\ {+}{=}\ \delta\vec{N}_X;$

        $u\ {+}{=}\ C_{11};\ v\ {+}{=}\ C_{12};$

    endfor

    $X_{\text{start}}\ {+}{=}\ \delta X_Y^s;\ X_{\text{end}}\ {+}{=}\ \delta X_Y^e;\ \vec{N}_{\text{start}}\ {+}{=}\ \delta\vec{N}_Y^s;$

    $u_s\ {+}{=}\ \delta u_Y^s;\ v_s\ {+}{=}\ \delta v_Y^s;$

endfor

Texture order polygon texturing

The inherent problem of texture order methods — i.e., that the uniform sampling of texture space does not guarantee the uniform sampling of screen space, and therefore may leave holes in the image or may generate pixels redundantly in an unexpected way — does not exist if the entire texture mapping is linear. Thus, we shall consider the simplified case when triangles are parameterized and orthographic projection is used, producing a linear texturing transformation.

The isoparametric lines of the texture space may be rotated and scaled by the texturing transformation, requiring the shaded polygons to be filled by possibly diagonal lines. Suppose the texture is defined by an $N \times M$ array T of "texture space pixels" or **texels**, and the complete texturing transformation is linear and has the following form:

$$X(u, v) = c_{11} \cdot u + c_{21} \cdot v + c_{31}, \qquad Y(u, v) = c_{12} \cdot u + c_{22} \cdot v + c_{32}. \quad (12.30)$$

The triangle being textured is defined both in 2D texture space and in the 3D screen space (figure 12.4). In texture order methods those texels must be found which cover the texture space triangle, and the corresponding screen space pixels should be shaded using the parameters found in the given location of the texture array. The determination of the covering texels in the texture space triangle is, in fact, a two-dimensional polygon-fill problem that has straightforward algorithms. The direct application of these algorithms, however, cannot circumvent the problem of different texture and pixel sampling grids, and thus can produce holes and overlapping in screen space. The complete isoparametric line of v from $u = 0$ to $u = 1$ is
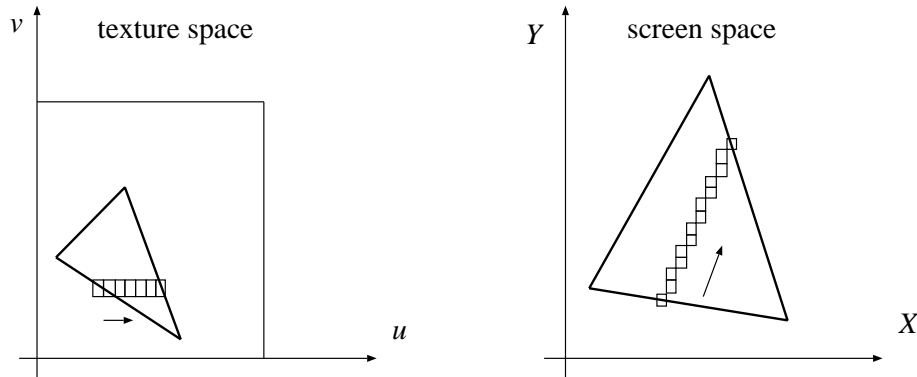
Figure 12.4: Texture order mapping

a digital line consisting of $d_u = \text{Trunc}(\max[c_{11}, c_{12}])$ pixels in screen space. Thus, the N number of texture pixels should be re-distributed to d_u screen pixels, which is equivalent to obtaining the texture array in $\delta u = d_u/N$ steps. Note that δu is not necessarily an integer, requiring a non-integer U coordinate which accumulates the δu increments. The integer index to the texture array is defined by the integer part of this U value. Nevertheless, U and δu can be represented in a fixed point format and calculated only by fixed point additions. Note that the emerging algorithm is similar to incremental line drawing methods, and eventually the distribution of N texture pixels onto d_u screen pixels is equivalent to drawing an incremental line with a slope N/d_u.

The complete isoparametric lines of u from $v = 0$ to $v = 1$ are similarly $d_v = \mathrm{Trunc}(\max[c_{21}, c_{22}])$ pixel long digital lines in screen space. The required increment of the v coordinate is $\delta v = d_v/M$ when the next pixel is generated on this digital line. Thus, a modified polygon-filling algorithm must be used in texture space to generate the texture values of the inner points, and it must be one which moves along the horizontal and vertical axes at d_u and d_v increments instead of jumping to integer points as in normal filling.
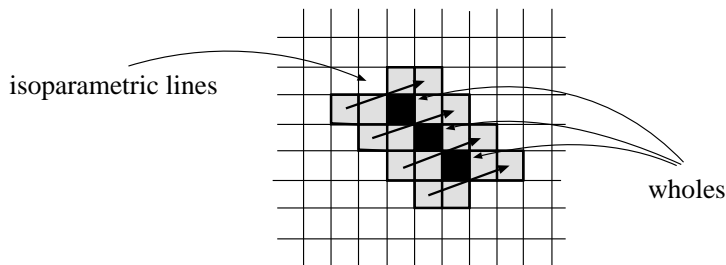
Figure 12.5: Holes between adjacent lines

This incremental technique must be combined with the filling of the screen space triangle. Since the isoparametric lines corresponding to constant v values are generally not horizontal but can have any slant, the necessary filling algorithm produces the internal pixels as a sequence of diagonal span lines. As in normal filling algorithms, two line generators are needed to produce the start and end points of the internal spans. A third line generator, on the other hand, generates the pixels between the start and end points.

The main problem of this approach is that using all pixels along the two parallel edges does not guarantee that all internal pixels will be covered by the connecting digital lines. Holes may appear between adjacent lines as shown in figure 12.5. Even linear texture transformation fails to avoid generating holes, but at least it does it in a well defined manner. Braccini and Marino [BG86] proposed drawing an extra pixel at each bend in the pixel space digital line to fill in any gaps that might be present. This is obviously a drastic approach and may result in redundancy, but it solves the inherent problem of texture order methods.

12.3.3 Texture mapping in the radiosity method

The general radiosity method consists of two steps: a view-independent radiosity calculation step, and a view-dependent rendering step where either an incremental shading technique is used, such as Gouraud shading, or else the shading is accomplished by ray-tracing. During the radiosity calculation step, the surfaces are broken down into planar elemental polygons which are assumed to have uniform radiosity, emission and diffuse coefficients. These assumptions can be made even if texture mapping is used, by calculating the "average" diffuse coefficient for each elemental surface, because the results of this approximation are usually acceptable. In the second, view-dependent step, however, textures can be handled as discussed for incremental shading and ray tracing.

12.4 Filtering of textures

Texture mapping establishes a correspondence between texture space and screen space. This mapping may magnify or shrink texture space regions when they are eventually projected onto pixels. Since raster based systems use regular sampling in pixel space, texture mapping can cause extremely uneven sampling of texture space, which inevitably results in strong aliasing artifacts. The methods discussed in chapter 11 (on sampling and quantization artifacts) must be applied to filter the texture in order to avoid aliasing. The applicable filtering techniques fall into two categories: pre-filtering, and post-filtering with supersampling.

12.4.1 Pre-filtering of textures

The difficulty of pre-filtering methods is that the mapping between texture and pixel spaces is usually non-linear, thus the shape of the convolution filter is distorted. The requirement for box filtering, for example, is the pre-image of a pixel, which is a curvilinear quadrilateral, and thus the texels lying inside this curvilinear quadrilateral must be summed to produce the pixel color. In order to ease the filtering computation, this curvilinear quadrilateral is approximated by some simpler geometric object; the possible alternatives are a square or a rectangle lying parallel to the texture coordinate axes, a normal quadrilateral or an ellipse (figure 12.6).
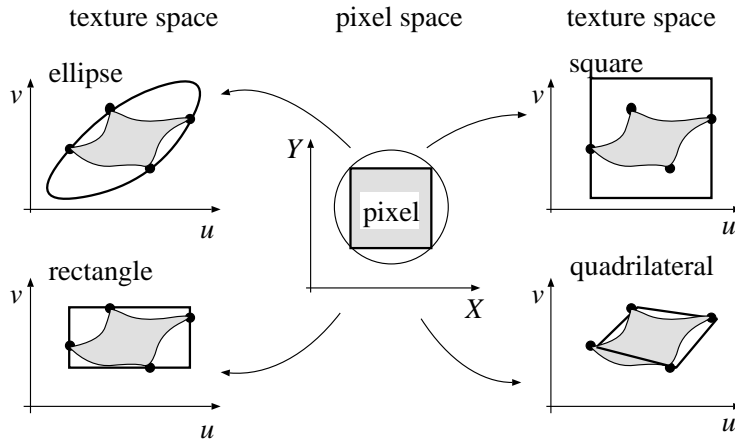
Figure 12.6: Approximations of the pre-image of the pixel rectangle

Rectangle approximation of the pre-image

The approximation by a rectangle is particularly suited to the Catmull texturing algorithm based on parallel subdivision in screen and texture space. Recall that in his method a patch is subdivided until the resulting polygon covers a single pixel. At the end of the process the corresponding texture domain subdivision takes the form of a square or a rectangle in texture space. Texels enclosed by this rectangle are added up, or the texture function is integrated here, to approximate a box filter. A conical or pyramidal filter can also be applied if the texels are weighted by a linear function increasing from the edges towards the center of the rectangle.

The calculation of the color of a single pixel requires integration or summation of the texels lying in its pre-image, and thus the computational burden is proportional to the size of the pre-image of the actual pixel. This can be disadvantageous if large textures are mapped onto a small area of the screen. Nevertheless, texture mapping can be speeded up, and this linear dependence on the size of the pixel's pre-image can be obviated if pre-integrated tables, or so-called **pyramids**, are used.

The image pyramid of pre-filtered data

Pyramids are multi-resolution data structures which contain the successively band-limited and subsampled versions of the same original image. These versions correspond to different sampling resolutions of the image. The resolution of the successive levels usually decrease by a factor of two. Conceptually, the subsequent images can be thought of as forming a pyramid, with the original image at the base and the crudest approximation at the apex, which provides some explanation of the name of this method. Versions are usually generated by box filtering and resampling the previous version of the image; that is, by averaging the color of four texels from one image, we arrive at the color of a texel for the subsequent level.
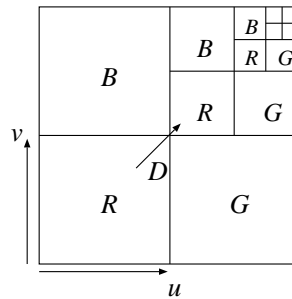
Figure 12.7: Mip-map organization of the memory

The collection of texture images can be organized into a **mip-map scheme**, as proposed by Williams [Wil83] (figure 12.7). ("mip" is an acronym of "multum in parvo" — "many things in a small space".) The mip-map scheme has a modest memory requirement. If the size of the original image is 1, then the cost of the mip-map organization is $1 + 2^{-2} + 2^{-4} + \ldots \approx 1.33$. The texture stored in a mip-map scheme is accessed using three indices: u, v texture coordinates and D for level of the pyramid. Looking up a texel defined by this three-index directory in the two-dimensional $M \times M$ mip-map array (MM) is a straightforward process:

$$R(u, v, D) = MM[(1 - 2^{-D}) \cdot M + u \cdot 2^{-D}, (1 - 2^{-D}) \cdot M + v \cdot 2^{-D}],$$
$$G(u, v, D) = MM[(1 - 2^{-(D+1)}) \cdot M + u \cdot 2^{-D}, (1 - 2^{-D}) \cdot M + v \cdot 2^{-D}],$$
$$B(u, v, D) = MM[(1 - 2^{-D}) \cdot M + u \cdot 2^{-D}, (1 - 2^{-(D+1)}) \cdot M + v \cdot 2^{-D}].$$
$$(12.31)$$

The application of the mip-map organization makes it possible to calculate the filtered color of the pixel in constant time and independently of the number of texels involved if D is selected appropriately. Level parameter D must obviously be derived from the span of the pre-image of the pixel area, d, which can be approximated as follows:

$$d = \max\{|[u(x+1), v(x+1)] - [u(x), v(x)]|, |[u(y+1), v(y+1)] - [u(y), v(y)]|\}$$

$$\approx \max\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}\}. \qquad (12.32)$$

The appropriate image version is that which composes approximately d^2 pixels together, thus the required pyramid level is:

$$D = \log_2(\max\{d, 1\}). \qquad (12.33)$$

The minimum 1 value in the above equation is justified by the fact that if the inverse texture mapping maps a pixel onto a part of a texel, then no filtering is necessary. The resulting D parameter is a continuous value which must be made discrete in order to generate an index for accessing the mip-map array. Simple truncation or rounding might result in discontinuities where the span size of the pixel pre-image changes, and thus would require some inter-level blending or interpolation. Linear interpolation is suitable for this task, thus the final expression of color values is:

$$\begin{aligned} &R(u, v, \text{Trunc}(D)) \cdot (1 - \text{Fract}(D)) + R(u, v, \text{Trunc}(D) + 1) \cdot \text{Fract}(D), \\ &G(u, v, \text{Trunc}(D)) \cdot (1 - \text{Fract}(D)) + G(u, v, \text{Trunc}(D) + 1) \cdot \text{Fract}(D), \\ &B(u, v, \text{Trunc}(D)) \cdot (1 - \text{Fract}(D)) + B(u, v, \text{Trunc}(D) + 1) \cdot \text{Fract}(D). \end{aligned}$$
$$(12.34)$$

The image pyramid relies on the assumption that the pre-image of the pixel can be approximated by a square in texture space. An alternative discussed by the next section, however, allows for rectangular areas oriented parallel to the coordinate axes.

Summed-area tables

A **summed-area table** (SA) is an array-like data structure which contains the running sum of colors as the image is scanned along successive scanlines; that is, at $[i, j]$ position of this SA table there is a triple of R, G, B

values, each of them generated from the respective T texel array as follows:

$$SA_I[i,j] = \sum_{u=0}^{i} \sum_{v=0}^{j} T[u,v]_I \tag{12.35}$$

where the subscript I stands for any of R, G or B.

This data structure makes it possible to calculate the box filtered or area summed value of any rectangle oriented parallel to the axes, since the sum of pixel colors in a rectangle given by corner points $[u_0, v_0; u_1, v_1]$ is:

$$I([u_0, v_0; u_1, v_1]) = \sum_{u=u_0}^{u_1} \sum_{v=v_0}^{v_1} T[u,v]_I =$$

$$SA_I[u_1, v_1] - SA_I[u_1, v_0] - SA_I[u_0, v_1] + SA_I[u_0, v_0]. \tag{12.36}$$

Image pyramids and summed-area tables allow for constant time filtering, but require set-up overhead to build the data structure. Thus they are suitable for textures which are used many times.

Rectangle approximation of the pixel's pre-image

Deeming the curvilinear region to be a normal quadrilateral provides a more accurate method [Cat74]. Theoretically the generation of internal texels poses no problem, because polygon filling algorithms are effective tools for this, but the implementation is not so simple as for a square parallel to the axes. Pyramidal filters are also available if an appropriate weighting function is applied [BN76].

EWA — Elliptical Weighted Average

Gangnet, Perny and Coueignoux [GPC82] came up with an interesting idea which considers pixels as circles rather than squares. The pre-image of a pixel is then an ellipse even for homogeneous transformations, or else can be quite well approximated by an ellipse even for arbitrary transformations. Ellipses thus form a uniform class of pixel pre-images, which can conveniently be represented by few parameters. The idea has been further refined by Greene and Heckbert [GH86] who proposed distorting the filter kernel according to the resulting ellipse in texture space.

expression of F. Thus, the $f(u, v)$ value can be directly used to generate the weighting factor of the selected filter. For a cone filter, for example, the weighting function is:

$$w(f) = \frac{\sqrt{f(u, v)}}{|u_x v_y - u_y v_x|}.$$ (12.41)

The square root compensates for the square of r in the expression of F. Apart from for cone filters, almost any kind of filter kernel (Gaussian, B-spline, sinc etc.) can be realized in this way, making the EWA approach a versatile and effective technique.

12.4.2 Post-filtering of textures

Post-filtering combined with supersampling means calculating the image at a higher resolution. The pixel colors are then computed by averaging the colors of subpixels belonging to any given pixel. The determination of the necessary subpixel resolution poses a critical problem when texture mapped surfaces are present, because it depends on both the texel resolution and the size of the areas mapped onto a single pixel, that is on the level of compression of the texturing, modeling and viewing transformations. By breaking down a pixel into a given number of subpixels, it is still not guaranteed that the corresponding texture space sampling will meet, at least approximately, the requirements of the sampling theorem. Clearly, the level of pixel subdivision must be determined by examination of all the factors involved. An interesting solution is given to this problem in the *REYES Image Rendering System* [CCC87] (REYES stands for "Renders Everything You Ever Saw"), where supersampling has been combined with stochastic sampling. In REYES surfaces are broken down into so-called **micropolygons** such that a micropolygon will have the size of about half a pixel when it goes through the transformations of image synthesis. Micropolygons have constant color determined by evaluation of the shading equation for coefficients coming from pre-filtered textures. Thus, at micropolygon level, the system applies a pre-filtering strategy. Constant color micropolygons are projected onto the pixel space, where at each pixel several subpixels are placed randomly, and the pixel color is computed by averaging those subpixels. Hence, on surface level a stochastic supersampling approach is used with post-filtering. The adaptivity of the whole method is provided

by the subdivision criterion of micropolygons; that is, they must have approximately half a pixel size after projection. The half pixel size is in fact the Nyquist limit of sampling on the pixel grid.

12.5 Bump mapping

Examining the formulae of shading equations we can see that the surface normal plays a crucial part in the computation of the color of the surface. Bumpy surfaces, such as the moon with its craters, have darker and brighter patches on them, since the modified normal of bumps can turn towards or away from the lightsources. Suppose that the image of a slightly bumpy surface has to be generated, where the height of the bumps is considerably smaller than the size of the object. The development of a geometric model to represent the surface and its bumps would be an algebraic nightmare, not to mention the difficulties of the generation of its image. Fortunately, we can apply a deft and convenient approximation method called **bump mapping**. The geometry used in transformations and visibility calculations is not intended to take the bumps into account — the moon, for example, is assumed to be a sphere — but during shading calculations a perturbed normal vector, taking into account the geometry and the bumps as well, is used in the shading equation. The necessary perturbation function is stored in texture maps, called bump maps, making bump mapping a special type of texture mapping. An appropriate perturbation of the normal vector gives the illusion of small valleys, providing the expected image without the computational burden of the geometry of the bumps. Now the derivation of the perturbations of the normal vectors is discussed, based on the work of Blinn [Bli78].

Suppose that the surface incorporating bumps is defined by a function $\vec{r}(u, v)$, and its smooth approximation is defined by $\vec{s}(u, v)$, that is, $\vec{r}(u, v)$ can be expressed by adding a small displacement $d(u, v)$ to the surface $\vec{s}(u, v)$ in the direction of its surface normal (figure 12.8). Since the surface normal $\vec{n}_s$ of $\vec{s}(u, v)$ can be expressed as the cross product of the partial derivatives $(\vec{s}_u, \vec{s}_v)$ of the surface in two parameter directions, we can write:

$$\vec{r}(u, v) = \vec{s}(u, v) + d(u, v) \cdot [\vec{s}_u(u, v) \times \vec{s}_v(u, v)]^0 = \vec{s}(u, v) + d(u, v) \cdot \vec{n}_s^0 \quad (12.42)$$

(the 0 superscript stands for unit vectors).
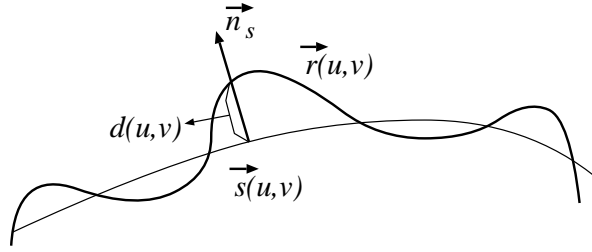
Figure 12.8: Description of bumps

The partial derivatives of $\vec{r}(u, v)$ are:

$$\vec{r}_u = \vec{s}_u + d_u \cdot \vec{n}_s^0 + d \cdot \frac{\partial \vec{n}_s^0}{\partial u},$$

$$\vec{r}_v = \vec{s}_v + d_v \cdot \vec{n}_s^0 + d \cdot \frac{\partial \vec{n}_s^0}{\partial v}. \tag{12.43}$$

The last terms can be ignored, since the normal vector variation is small for smooth surfaces, as is the $d(u, v)$ bump displacement, thus:

$$\vec{r}_u \approx \vec{s}_u + d_u \cdot \vec{n}_s^0,$$

$$\vec{r}_v \approx \vec{s}_v + d_v \cdot \vec{n}_s^0. \tag{12.44}$$

The surface normal of $r(u, v)$, that is the perturbed normal, is then:

$$\vec{n}_r = \vec{r}_u \times \vec{r}_v = \vec{s}_u \times \vec{s}_v + d_u \cdot \vec{n}_s^0 \times \vec{s}_v + d_v \cdot \vec{s}_u \times \vec{n}_s^0 + d_u d_v \cdot \vec{n}_s^0 \times \vec{n}_s^0. \tag{12.45}$$

Since the last term of this expression is identically zero because of the axioms of the vector product, and

$$\vec{n}_s = \vec{s}_u \times \vec{s}_v, \qquad \vec{s}_u \times \vec{n}_s^0 = -\vec{n}_s^0 \times \vec{s}_u,$$

thus we get:

$$\vec{n}_r = \vec{n}_s + d_u \cdot \vec{n}_s^0 \times \vec{s}_v - d_v \cdot \vec{n}_s^0 \times \vec{s}_u. \tag{12.46}$$

This formula allows for the computation of the perturbed normal using the derivatives of the displacement function. The displacement function $d(u, v)$ must be defined by similar techniques as for texture maps; they

can be given either by functions or by pre-computed arrays called **bump maps**. Each time a normal vector is needed, the (u, v) coordinates have to be determined, and then the derivatives of the displacement function must be evaluated and substituted into the formula of the perturbation vector.

The formula of the perturbated vector requires the derivatives of the bump displacement function, not its value. Thus, we can either store the derivatives of the displacement function in two bump maps, or calculate them each time using finite differences. Suppose the displacement function is defined by an $N \times N$ bump map array, B. The calculated derivatives are then:

$$U = \mathrm{Trunc}(u * N); \qquad V = \mathrm{Trunc}(v * N);$$
if $U < 1$ **then** $U = 1$; **if** $U > N - 2$ **then** $U = N - 2$;
if $V < 1$ **then** $V = 1$; **if** $V > N - 2$ **then** $V = N - 2$;
$$d_u(u, v) = (B[U + 1, V] - B[U - 1, V]) \cdot N/2;$$
$$d_v(u, v) = (B[U, V + 1] - B[U, V - 1]) \cdot N/2;$$

The displacement function, $d(u, v)$, can be derived from frame-grabbed photos or hand-drawn digital pictures generated by painting programs, assuming color information to be depth values, or from z-buffer memory values of computer synthesized images. With the latter method, an arbitrary arrangement of 3D objects can be used for definition of the displacement of the bump-mapped surface.

Blinn [Bli78] has noticed that bump maps defined in this way are not invariant to scaling of the object. Suppose two differently scaled objects with the same bump map are displayed on the screen. One might expect the bigger object to have bigger wrinkles proportionally to the size of the object, but that will not be the case, since

$$\frac{|\vec{n}_r - \vec{n}_s|}{|\vec{n}_s|} = \frac{|d_u \cdot \vec{n}_s^0 \times \vec{s}_v - d_v \cdot \vec{n}_s^0 \times \vec{s}_u|}{|\vec{s}_u \times \vec{s}_v|}$$

is not invariant with the scaling of $\vec{s}(u, v)$ and consequently of $\vec{n}_s$, but it is actually inversely proportional to it. If it generates unwanted effects, then a compensation is needed for eliminating this dependence.

12.5.1 Filtering for bump mapping

The same problems may arise in the context of bump mapping as for texture mapping if point sampling is used for image generation. The applicable solution methods are also similar and include pre-filtering and post-filtering with supersampling. The pre-filtering technique — that is, the averaging of displacement values stored in the bump map — contains, however, the theoretical difficulty that the dependence of surface colors on bump displacements is strongly non-linear. In effect, pre-filtering will tend to smooth out not only high-frequency aliases, but also bumps.

12.6 Reflection mapping

Reading the section on bump mapping, we can see that texture mapping is a tool which can re-introduce features which were previously eliminated because of algorithmic complexity considerations. The description of bumps by their true geometric characteristics is prohibitively expensive computationally, but this special type of texture mapping, bump mapping, can provide nearly the same effect without increasing the geometric complexity of the model. Thus it is no surprise that attempts have been made to deal with other otherwise complex phenomena within the framework of texture mapping. The most important class of these approaches addresses the problem of coherent reflection which could otherwise be solved only by expensive ray tracing.

A reflective object reflects the image of its environment into the direction of the camera. Thus, the pre-computed image visible from the center of the reflective object can be used later, when the color visible from the camera is calculated. These pre-computed images with respect to reflective objects are called **reflection maps** [BN76]. Originally Blinn and Newell proposed a sphere as an intermediate object onto which the environment is projected. Cubes, however, are more convenient [MH84], since to generate the pictures seen through the six sides of the cube, the same methods can be used as for computing the normal images from the camera.

Having generated the images visible from the center of reflective objects, a normal image synthesis algorithm can be started using the real camera. Reflective surfaces are treated as texture mapped ones with the texture
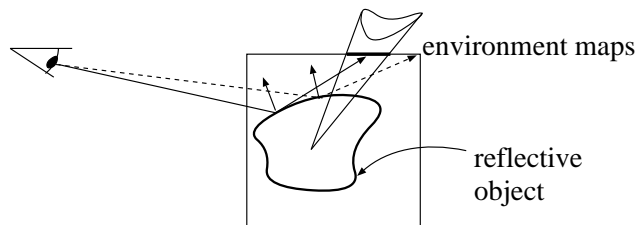
Figure 12.9: Reflection mapping

coordinates calculated by taking into account not only the surface point, but the viewing direction and the surface normal as well. By reflecting the viewing direction — that is, the vector pointing to the surface point from the camera — onto the surface normal, the reflection direction vector is derived, which unambiguously defines a point on the intermediate cube (or sphere), which must be used to provide the color of the reflection direction. This generally requires a cube-ray intersection. However, if the cube is significantly greater than the object itself, the dependence on the surface point can be ignored, which allows for the access of the environment map by the coordinates of the reflection vector only.

Suppose the cube is oriented parallel to the coordinate axes and the images visible from the center of the object through its six sides are stored in texture maps $R[0, u, v], R[1, u, v], \ldots, R[5, u, v]$. Let the reflected view vector be $\vec{V_r} = [V_x, V_y, V_z]$.

The *color* of the light coming from the reflection direction is then:

$$\begin{aligned}
&\textbf{if } |V_x| = \max\{|V_x|, |V_y|, |V_z|\} \textbf{ then}\\
&\quad \textbf{if } V_x > 0 \textbf{ then } \text{color} = R[0, 0.5 + V_y/V_x, 0.5 + V_z/V_x];\\
&\quad \textbf{else} \qquad\qquad \text{color} = R[3, 0.5 + V_y/V_x, 0.5 + V_z/V_x];\\
&\textbf{if } |V_y| = \max\{|V_x|, |V_y|, |V_z|\} \textbf{ then}\\
&\quad \textbf{if } V_y > 0 \textbf{ then } \text{color} = R[1, 0.5 + V_x/V_y, 0.5 + V_z/V_y];\\
&\quad \textbf{else} \qquad\qquad \text{color} = R[4, 0.5 + V_x/V_y, 0.5 + V_z/V_y];\\
&\textbf{if } |V_z| = \max\{|V_x|, |V_y|, |V_z|\} \textbf{ then}\\
&\quad \textbf{if } V_z > 0 \textbf{ then } \text{color} = R[2, 0.5 + V_x/V_z, 0.5 + V_y/V_z];\\
&\quad \textbf{else} \qquad\qquad \text{color} = R[5, 0.5 + V_x/V_z, 0.5 + V_y/V_z];
\end{aligned}$$

Chapter 13

ANIMATION

Animation introduces time-varying features into computer graphics, most importantly the motion of objects and the camera. Theoretically, all the parameters defining a scene in the virtual world model can be functions of time, including color, size, shape, optical coefficients, etc., but these are rarely animated, thus we shall mainly concentrate on motion and camera animation. In order to illustrate motion and other time-varying phenomena, animation systems generate not a single image of a virtual world, but a sequence of them, where each image represents a different point of time. The images are shown one by one on the screen allowing a short time for the user to have a look at each one before the next is displayed.

Supposing that the objects are defined in their respective local coordinate system, the position and orientation of the particular object is given by its modeling transformation. Recall that this modeling transformation places the objects in the global world coordinate system determining the relative position and orientation from other objects and the camera.

The camera parameters, on the other hand, which include the position and orientation of the 3D window and the relative location of the camera, are given in the global coordinate system thus defining the viewing transformation which takes us from the world to the screen coordinate system.

Both transformations can be characterized by 4×4 homogeneous matrices. Let the time-varying modeling transformation of object o be $\mathbf{T}_{\mathbf{M,o}}(t)$ and the viewing transformation be $\mathbf{T}_{\mathbf{V}}(t)$.

A simplistic algorithm of the generation of an animation sequence, assuming a built-in timer, is:

Initialize Timer(t_start);
do
    t = Read Timer;
    for each object o **do** Set modeling transformation: $\mathbf{T}_{\mathbf{M,o}} = \mathbf{T}_{\mathbf{M,o}}(t)$;
    Set viewing transformation: $\mathbf{T_V} = \mathbf{T_V}(t)$;
    Generate Image;
while $t < t_\text{end}$;

In order to provide the effect of continuous motion, a new static image should be generated at least every 60 msec. If the computer is capable of producing the sequence at such a speed, we call this **real-time animation**, since now the timer can provide real time values. With less powerful computers we are still able to generate continuous looking sequences by storing the computed image sequence on mass storage, such as a video recorder, and replaying them later. This technique, called **non-real-time animation**, requires the calculation of subsequent images in uniformly distributed time samples. The time gap between the samples has to exceed the load time of the image from the mass storage, and should meet the requirements of continuous motion as well. The general sequence of this type of animation is:

$t = t_\text{start}$;   *// preprocessing phase: recording*
do
    for each object o **do** Set modeling transformation: $\mathbf{T}_{\mathbf{M,o}} = \mathbf{T}_{\mathbf{M,o}}(t)$;
    Set viewing transformation: $\mathbf{T_V} = \mathbf{T_V}(t)$;
    Generate Image;
    Store Image;
    $t \mathrel{+}= \Delta t$;
while $t < t_\text{end}$;
Initialize Timer(t_start) ;   *// animation phase: replay*
do
    t = Read Timer;
    Load next image;
    $t \mathrel{+}= \Delta t$;
    while ($t >$ Read Timer) Wait;
while $t < t_\text{end}$;

Note that the only responsibility of the animation phase is the loading and the display of the subsequent images at each time interval Δt. The simplest way to do this is to use a commercial VCR and television set, having recorded the images computed for arbitrary time on a video tape frame-by-frame in analog signal form. In this way computers are used only for the preprocessing phase, the real-time display of the animation sequence is generated by other equipments developed for this purpose.

As in traditional computer graphics, the objective of animation is to generate realistic motion. Motion is realistic if it is similar to what observers are used to in their everyday lives. The motion of natural objects obeys the laws of physics, specifically, **Newton's law** stating that the acceleration of masses is proportional to the resultant driving force. Let a point of object mass have positional vector $\vec{r}(t)$ at time t, and assume that the resultant driving force on this mass is $\vec{D}$. The position vector can be expressed by the modeling transformation and the position in the local coordinates ($\vec{r}_L$):

$$\vec{r}(t) = \vec{r}_L \cdot \mathbf{T_M}(t). \tag{13.1}$$
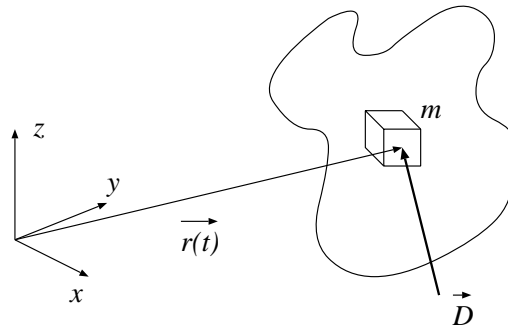
Figure 13.1: Dynamics of a single point of mass in an object

Newton's law expresses the second derivative (acceleration) of $\vec{r}(t)$ using the driving force $\vec{D}$ and the mass m:

$$\frac{\vec{D}}{m} = \frac{d^2\vec{r}(t)}{dt^2} = \vec{r}_L \cdot \frac{d^2\mathbf{T_M}(t)}{dt^2}. \tag{13.2}$$

Since there are only finite forces in nature, the second derivative of all elements of the transformation matrix must be finite. More precisely, the second derivative has to be continuous, since mechanical forces cannot change abruptly, because they act on some elastic mechanism, making $\vec{D}(t)$ continuous. To summarize, the illusion of realistic motion requires the elements of the transformation matrices to have finite and continuous second derivatives. Functions meeting these requirements belong to the C^2 family (C stands for parametric continuity, and superscript 2 denotes that the second derivatives are regarded). The C^2 property implies that the function is also of type C^1 and C^0.

The crucial problem of animation is the definition of the appropriate matrices $\mathbf{T_M}(t)$ and $\mathbf{T_V}(t)$ to reflect user intention and also to give the illusion of realistic motion. This task is called **motion control**.

To allow maximum flexibility, interpolation and approximation techniques applied in the design of free form curves are recommended here. The designer of the motion defines the position and the orientation of the objects just at a few knot points of time, and the computer interpolates or approximates a function depending on these knot points taking into account the requirements of realistic motion. The interpolated function is then sampled in points required by the animation loop.
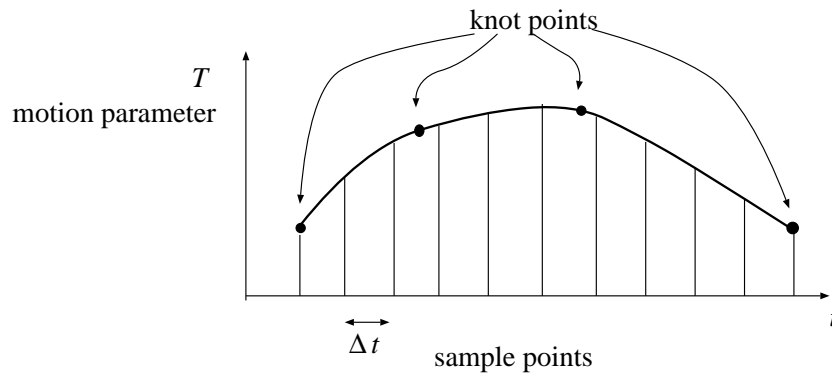
Figure 13.2: Motion control by interpolation

13.1 Interpolation of position-orientation matrices

As we know, arbitrary position and orientation can be defined by a matrix of the following form:

$$
\begin{bmatrix}
 & & 0 \\
\mathbf{A}_{3\times 3} & & 0 \\
 & & 0 \\
\mathbf{q^T} & & 1
\end{bmatrix}
=
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & 0 \\
a_{21} & a_{22} & a_{23} & 0 \\
a_{31} & a_{32} & a_{33} & 0 \\
q_x & q_y & q_z & 1
\end{bmatrix}.
\tag{13.3}
$$

Vector $\mathbf{q^T}$ sets the position and $\mathbf{A}_{3\times 3}$ is responsible for defining the orientation of the object. The elements of $\mathbf{q^T}$ can be controlled independently, adjusting the x, y and z coordinates of the position. Matrix elements $a_{11}, \ldots a_{33}$, however, are not independent, since the degree of freedom in orientation is 3, not 9, the number of elements in the matrix. In fact, a matrix representing a valid orientation must not change the size and the shape of the object, thus requiring the row vectors of $\mathbf{A}$ to be unit vectors forming a perpendicular triple. Matrices having this property are called **orthonormal**.

Concerning the interpolation, the elements of the position vector can be interpolated independently, but independent interpolation is not permitted for the elements of the orientation matrix, since the interpolated elements would make non-orthonormal matrices. A possible solution to this problem is to interpolate in the space of the roll/pitch/yaw (α, β, γ) angles (see section 5.1.1), since they form a basis in the space of the orientations, that is, any roll-pitch-yaw triple represents an orientation, and all orientations can be expressed in this way. Consequently, the time functions describing the motion are:

$$
\mathbf{p}(t) = [x(t), y(t), z(t), \alpha(t), \beta(t), \gamma(t)]
\tag{13.4}
$$

($\mathbf{p}(t)$ is called parameter vector).

In image generation the homogeneous matrix form of transformation is needed, and thus, having calculated the position value and orientation angles, the transformation matrix has to be expressed.

Using the definitions of the roll, pitch and yaw angles:

$$
\mathbf{A} = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & \sin\gamma \\ 0 & -\sin\gamma & \cos\gamma \end{bmatrix}.
$$
$$(13.5)$$

The position vector is obviously:

$$\mathbf{q^T} = [x, y, z]. \tag{13.6}$$

We concluded that in order to generate realistic motion, $\mathbf{T}$ should have continuous second derivatives. The interpolation is executed, however, in the space of the parameter vectors, meaning that this requirement must be replaced by another one concerning the position values and orientation angles.

The modeling transformation depends on time indirectly, through the parameter vector:

$$\mathbf{T} = \mathbf{T}(\mathbf{p}(t)). \tag{13.7}$$

Expressing the second derivative of a matrix element T_{ij}:

$$\frac{dT_{ij}}{dt} = grad_{\mathbf{p}}T_{ij} \cdot \dot{\mathbf{p}}, \tag{13.8}$$

$$\frac{d^2 T_{ij}}{dt^2} = (\frac{d}{dt}\mathrm{grad}_{\mathbf{p}}T_{ij}) \cdot \dot{\mathbf{p}} + \mathrm{grad}_{\mathbf{p}}T_{ij} \cdot \ddot{\mathbf{p}} = h(\mathbf{p}, \dot{\mathbf{p}}) + H(\mathbf{p}) \cdot \ddot{\mathbf{p}}. \tag{13.9}$$

In order to guarantee that d^2T_{ij}/dt^2 is finite and continuous, $\ddot{\mathbf{p}}$ should also be finite and continuous. This means that the interpolation method used for the elements of the parameter vector has to generate functions which have continuous second derivatives, or in other words, has to provide C^2 continuity.

There are several alternative interpolation methods which satisfy the above criterion. We shall consider a very popular method which is based on cubic B-splines.

13.2 Interpolation of the camera parameters

Interpolation along the path of the camera is a little bit more difficult, since a complete camera setting contains more parameters than a position and orientation. Recall that the setting has been defined by the following independent values:

1. $v\vec{r}p$, view reference point, defining the center of the window,

2. $v\vec{p}n$, view plane normal, concerning the normal of the plane of the window,

3. $v\vec{u}p$, view up vector, showing the directions of the edges of the window,

4. $wheight, wwidth$, horizontal and vertical sizes of the window,

5. $e\vec{y}e$, the location of the camera in the u, v, w coordinate system fixed to the center of the window,

6. fp, bp, front and back clipping plane,

7. Type of projection.

Theoretically all of these parameters can be interpolated, except for the type of projection. The position of clipping planes, however, is not often varied with time, since clipping planes do not correspond to any natural phenomena, but are used to avoid overflows in the computations and to simplify the projection.

Some of the above parameters are not completely independent. Vectors $v\vec{p}n$ and $v\vec{u}p$ ought to be perpendicular unit vectors. Fortunately, the algorithm generating the viewing transformation matrix $\mathbf{T_V}$ takes care of these requirements, and should the two vectors not be perpendicular or of unit length, it adjusts them. Consequently, an appropriate space of independently adjustable parameters is:

$$\mathbf{p_{cam}}(t) = [v\vec{r}p, v\vec{p}n, v\vec{u}p, wheight, wwidth, e\vec{y}e, fp, bp]. \qquad (13.10)$$

As has been discussed in chapter 5 (on transformations, clipping and projection), these parameters define a viewing transformation matrix $\mathbf{T_V}$ if the following conditions hold:

$$v\vec{p}n \times v\vec{u}p \neq 0, \quad wheight \geq 0, \quad wwidth \geq 0, \quad eye_w < 0, \quad eye_w < fp < bp. \tag{13.11}$$

This means that the interpolation method has to take account not only of the existence of continuous derivatives of these parameters, but also of the requirements of the above inequalities for any possible point of time. In practical cases, the above conditions are checked in the knot points (keyframes) only, and then animation is attempted. Should it be that the path fails to provide these conditions, the animation system will then require the designer to modify the keyframes accordingly.

13.3   Motion design

The design of the animation sequence starts by specifying the knot points of the interpolation. The designer sets several time points, say $t_1, t_2 \ldots t_n$, and defines the position and orientation of objects and the camera in these points of time. This information could be expressed in the form of the parameter vector, or of the transformation matrix. In the latter case, the parameter vector should be derived from the transformation matrix for the interpolation. This task, called the **inverse geometric problem**, involves the solution of several trigonometric equations and is quite straightforward due to the formulation of the orientation matrix based on the roll, pitch and yaw angles.

Arranging the objects in t_i, we define a knot point for a parameter vector $\mathbf{p_o}(t_i)$ for each object o. These knot points will be used to interpolate a C^2 function (e.g. a B-spline) for each parameter of each object, completing the design phase.

In the animation phase, the parameter functions are sampled in the actual time instant, and the respective transformation matrices are set. Then the image is generated.

These steps are summarized in the following algorithm:

Define the time of knot points: $t_1, \ldots, t_n$;   // *design phase*
for each knot point k **do**
    for each object o **do**
        Arrange object o:
$$\mathbf{p_o}(t_k) = [x(t_k), y(t_k), z(t_k), \alpha(t_k), \beta(t_k), \gamma(t_k)]_o;$$
    endfor
    Set camera parameters: $\mathbf{p_{cam}}(t_k)$;
endfor
for each object o **do**
    Interpolate a C^2 function for:
$$\mathbf{p_o}(t) = [x(t), y(t), z(t), \alpha(t), \beta(t), \gamma(t)]_o;$$
endfor
Interpolate a C^2 function for: $\mathbf{p_{cam}}(t)$;

Initialize Timer(t_{start});   // *animation phase*
do
    t = Read Timer;
    for each object o **do**
        Sample parameters of object o:
$$\mathbf{p_o} = [x(t), y(t), z(t), \alpha(t), \beta(t), \gamma(t)]_o;$$
      $\mathbf{T_{M,o}} = \mathbf{T_{M,o}}(\mathbf{p_o})$;
    endfor
    Sample parameters of camera: $\mathbf{p_{cam}} = \mathbf{p_{cam}}(t)$;
    $\mathbf{T_V} = \mathbf{T_V}(\mathbf{p_{cam}})$;
    Generate Image;
while $t < t_{\text{end}}$;

This approach has several disadvantages. Suppose that having designed and animated a sequence, we are not satisfied with the result, because we find that a particular part of the film is too slow, and we want to speed it up. The only thing we can do is to re-start the motion design from scratch and re-define all the knot points. This seems unreasonable, since it was not our aim to change the path of the objects, but only to modify the **kinematics** of the motion. Unfortunately, in the above approach both the geometry of

the trajectories and the kinematics (that is the speed and acceleration along the trajectories) are specified by the same transformation matrices. That is why this approach does not allow for simple kinematic modification.

This problem is not a new one for actors and actresses in theaters, since a performance is very similar to an animation sequence, where the objects are the actors themselves. Assume a performance were directed in the same fashion as the above algorithm. The actors need to know the exact time when they are supposed to come on the stage. What would happen if the schedule were slightly modified, because of a small delay in the first part of the performance? Every single actor would have to be given a new schedule. That would be a nightmare, would it not. Fortunately, theaters do not work that way. Dramas are broken down into scenes. Actors are aware of the scene when they have to come on, not the time, and there is a special person, called a stage manager, who keeps an eye on the performance and informs the actors when they are due on (all of them at once). If there is a delay, or the timing has to be modified, only the stage manager's schedule has to change, the actors' schedules are unaffected. The geometry of the trajectory (movement of actors) and the kinematics (timing) has been successfully separated.

The very same approach can be applied to computer animation as well. Now the sequence is broken down into **frames** (this is the analogy of scenes), and the geometry of the trajectories is defined in terms of frames ($\mathcal{F}$), not in terms of time. The knot points of frames are called **keyframes**, and conveniently the first keyframe defines the arrangement at $\mathcal{F} = 1$, the second at $\mathcal{F} = 2$ etc. The kinematics (stage manager) is introduced to the system by defining the sequence of frames in terms of time, resulting in a function $\mathcal{F}(t)$.

Concerning the animation phase, in order to generate an image in time t, first $\mathcal{F}(t)$ is evaluated, then the result is used to calculate $\mathbf{T}(\mathbf{p}(\mathcal{F}))$ matrices. Tasks, such as modifying the timing of the sequence, or even reversing the whole animation, can be accomplished by the proper modification of the frame function $\mathcal{F}(t)$ without affecting the transformation matrices.

Now the transformation matrices are defined indirectly, through $\mathcal{F}$. Thus special attention is needed to guarantee the continuity of second derivatives
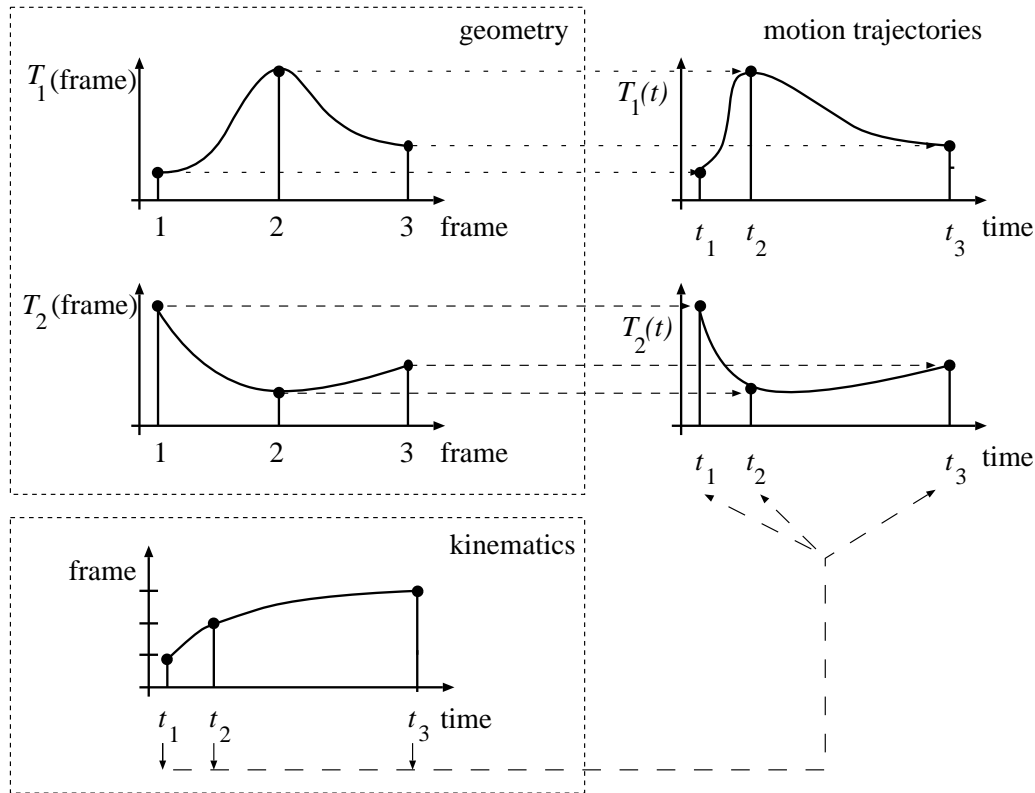
Figure 13.3: Keyframe animation

of the complete function. Expressing the second derivatives of T_{ij}, we get:

$$\frac{d^2 T_{ij}}{dt^2} = \frac{d^2 T_{ij}}{d\mathcal{F}^2} \cdot (\dot{\mathcal{F}})^2 + \frac{d T_{ij}}{d\mathcal{F}} \cdot \ddot{\mathcal{F}}. \qquad (13.12)$$

A sufficient condition for the continuous second derivatives of $T_{ij}(t)$ is that both $\mathcal{F}(t)$ and $T_{ij}(\mathcal{F})$ are C^2 functions. The latter condition requires that the parameter vector $\mathbf{p}$ is a C^2 type function of the frame variable $\mathcal{F}$.

The concept of **keyframe animation** is summarized in the following algorithm:

// *Design of the geometry*
for each keyframe kf **do**
    for each object o **do**
        Arrange object o:
$$\mathbf{p_o}(kf) = [x(kf), y(kf), z(kf), \alpha(kf), \beta(kf), \gamma(kf)]_o;$$
    endfor
    Set camera parameters: $\mathbf{p_{cam}}(kf)$;
endfor
for each object o **do**
    Interpolate a C^2 function for:
    $\mathbf{p_o}(f) = [x(f), y(f), z(f), \alpha(f), \beta(f), \gamma(f)]_o;$
endfor
Interpolate a C^2 function for: $\mathbf{p_{cam}}(f)$;

// *Design of the kinematics*
for each keyframe kf **do** Define t_{kf}, when $\mathcal{F}(t_{kf}) = kf$;
Interpolate a C^2 function for $\mathcal{F}(t)$;

// *Animation phase*
Initialize Timer(t_{start});
do
    $t = $ Read Timer;
    $f = \mathcal{F}(t)$;
    for each object o **do**
        Sample parameters of object o:
$$\mathbf{p_o} = [x(f), y(f), z(f), \alpha(f), \beta(f), \gamma(f)]_o;$$
        $\mathbf{T_{M,o}} = \mathbf{T_{M,o}}(\mathbf{p_o})$;
    endfor
    Sample parameters of camera: $\mathbf{p_{cam}}(f)$;
    $\mathbf{T_V} = \mathbf{T_V}(\mathbf{p_{cam}})$;
    Generate Image;
while $t < t_{\text{end}}$;

13.4 Parameter trajectory calculation

We concluded in the previous sections that the modeling ($\mathbf{T_M}$) and viewing ($\mathbf{T_V}$) transformation matrices should be controlled or animated via parameter vectors. Additionally, the elements of these parameter vectors and the frame function for keyframe animation must be C^2 functions — that is, they must have continuous second derivatives — to satisfy Newton's law which is essential if the motion is to be realistic. In fact, the second derivatives of these parameters are proportional to the force components, which must be continuous in real-life situations, and which in turn require the parameters to have this C^2 property.

We also stated that in order to allow for easy and flexible trajectory design, the designer of the motion defines the position and the orientation of the objects — that is indirectly the values of the motion parameters — in just a few knot points of time, and lets the computer interpolate a function relying on these knot points taking into account the requirements of C^2 continuity. The interpolated function is then sampled in points required by the animation loop.

This section focuses on this **interpolation** process which can be formulated for a single parameter as follows:

Suppose that points $[p_0, p_1, \ldots, p_n]$ are given with their respective time values $[t_0, t_1, \ldots, t_n]$, and we must find a function $p(t)$ that satisfies:

$$p(t_i) = p_i, \qquad i = 0, \ldots, n, \tag{13.13}$$

and that the second derivative of $p(t)$ is continuous (C^2 type). Function $p(t)$ must be easily represented and computed numerically, thus it is usually selected from the family of polynomials or the piecewise combination of polynomials. Whatever family is used, the number of its independently controllable parameters, called the degree of freedom, must be at least n, to allow the function to satisfy n number of independent equations.

Function $p(t)$ that satisfies equation 13.13 is called the **interpolation function** or curve of **knot** or **control points** $[t_0, p_0; t_1, p_1; \ldots; t_n, p_n]$. Interpolation functions are thus required to pass through its knot points. In many cases this is not an essential requirement, however, and it seems advantageous that a function should just follow the general directions provided by the knot points — that is, that it should only approximate the

knot points — if by thus eliminating the "passing through" constraint we improve other, more important, properties of the generated function. This latter function type is called the **approximation function**. This section considers interpolation functions first, then discusses the possibilities of approximation functions in computer animation.

A possible approach to interpolation can take a single, minimal order polynomial which satisfies the above criterion. The degree of the polynomial should be at least $n - 1$ to have n independent coefficients, thus the interpolation polynomial is:

$$p(t) = \sum_{i=0}^{n-1} a_i \cdot t^i. \tag{13.14}$$

The method that makes use of this approach is called **Lagrange interpolation**. It can be easily shown that the polynomial which is incident to the given knot points can be expressed as:

$$p(t) = \sum_{i=0}^{n} p_i \cdot L_i^{(n)}(t), \tag{13.15}$$

where $L_i^{(n)}(t)$, called the Lagrange base polynomial, is:

$$L_i^{(n)}(t) = \frac{\prod_{\substack{j=0 \\ j \neq i}}^{n} (t - t_j)}{\prod_{\substack{j=0 \\ j \neq i}}^{n} (t_i - t_j)}. \tag{13.16}$$

Equation 13.15 gives an interesting geometric interpretation to this schema. The Lagrange base polynomials are in fact **weight functions** which give a certain weight to the knot points in the linear combination defining $p(t)$. Thus, the value of $p(t)$ comes from the time-varying blend of the control points. The roots of blending functions $L_i^{(n)}(t)$ are $t_0, \ldots, t_{i-1}, t_{i+1}, \ldots t_n$, and the function gives positive or negative values in the subsequent ranges $[t_j, t_{j+1}]$, resulting in an oscillating shape. Due to the oscillating blending functions, the interpolated polynomial also tends to oscillate between even reasonably arranged knot points, thus the motion exhibits wild wiggles that are not inherent in the definition data. The greater the number of knot

points, the more noticeable the oscillations become since the degree of the base polynomials is one less that the number of knot points. Thus, although single polynomial based interpolation meets the requirement of continuous second derivatives and easy definition and calculation, it is acceptable for animation only if the degree, that is the number of knot points, is small.

A possible and promising direction of the refinement of the polynomial interpolation is the application of several polynomials of low degree instead of a single high-degree polynomial. This means that in order to interpolate through knot points $[t_0, p_0; t_1, p_1; \ldots; t_n, p_n]$, a different polynomial $p_i(t)$ is found for each range $[t_i, t_{i+1}]$ between the subsequent knot points. The complete interpolated function $p(t)$ will be the composition of the segment polynomials responsible for defining it in the different $[t_i, t_{i+1}]$ intervals, that is:

$$p(t) = \begin{cases} p_0(t) \text{ if } t_0 \leq t < t_1 \\ \vdots \\ p_i(t) \text{ if } t_i \leq t < t_{i+1} \\ \vdots \\ p_{n-1}(t) \text{ if } t_{n-1} \leq t \leq t_n \end{cases} \qquad (13.17)$$

In order to guarantee that $p(t)$ is a C^2 function, the segments must be carefully connected to provide C^2 continuity at the joints. Since this may mean different second derivatives required at the two endpoints of the segment, the polynomial must not have a constant second derivative, that is, at least cubic (3-degree) polynomials should be used for these segments. A composite function of different segments connected together to guarantee C^2 continuity is called a **spline** [RA89]. The simplest, but practically the most important, spline consists of 3-degree polynomials, and is therefore called the **cubic spline**. A cubic spline segment valid in $[t_i, t_{i+1}]$ can be written as:

$$p_i(t) = a_3 \cdot (t - t_i)^3 + a_2 \cdot (t - t_i)^2 + a_1 \cdot (t - t_i) + a_0. \qquad (13.18)$$

The coefficients (a_3, a_2, a_1, a_0) define the function unambiguously, but they cannot be given a direct geometrical interpretation. Thus an alternative representation is selected, which defines the values and the derivatives of the segment at the two endpoints, forming a quadruple $(p_i, p_{i+1}, p'_i, p'_{i+1})$. The correspondence between the coefficients of the polynomial can be established by calculating the values and the derivatives of equation 13.18.

Using the simplifying notation $T_i = t_{i+1} - t_i$, we get:

$$
\begin{aligned}
p_i &= p_i(t_i) = a_0, \\
p_{i+1} &= p_i(t_{i+1}) = a_3 \cdot T_i^3 + a_2 \cdot T_i^2 + a_1 \cdot T_i + a_0, \\
p_i' &= p_i'(t_i) = a_1, \\
p_{i+1}' &= p_i'(t_{i+1}) = 3a_3 \cdot T_i^2 + 2a_2 \cdot T_i + a_1.
\end{aligned}
\tag{13.19}
$$

These equations can be used to express $p_i(t)$ by the endpoint values and derivatives, proving that this is also an unambiguous representation:

$$
p_i(t) = [2(p_i - p_{i+1}) + (p_i' + p_{i+1}')T_i] \cdot \left(\frac{t - t_i}{T_i}\right)^3 +
$$

$$
[3(p_{i+1} - p_i) - (2p_i' + p_{i+1}')T_i] \cdot \left(\frac{t - t_i}{T_i}\right)^2 + p_i' \cdot (t - t_i) + p_i. \tag{13.20}
$$
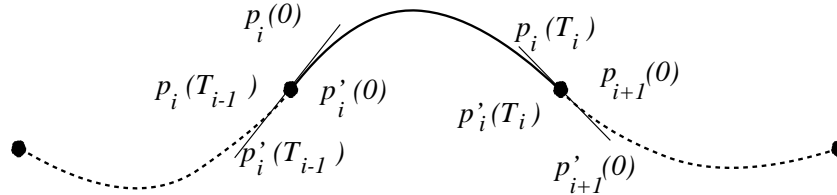
Figure 13.4: Cubic B-spline interpolation

The continuous connection of consecutive cubic segments expressed in this way is easy because C^0 and C^1 continuity is automatically provided if the value and the derivative of the endpoint of one segment is used as the starting point of the next segment. Only the continuity of the second derivative ($p''(t)$) must somehow be obtained. The first two elements in the quadruple $(p_i, p_{i+1}, p_i', p_{i+1}')$ are the knot points which are known before the calculation. The derivatives at these points, however, are usually not available, thus they must be determined from the requirement of C^2 continuous connection. Expressing the second derivative of the function defined by equation 13.20 for any k and $k+1$, and requiring $p_k''(t_{k+1}) = p_{k+1}''(t_{k+1})$, we get:

$$
T_{k+1}p_k' + 2(T_k + T_{k+1})p_{k+1}' + T_k p_{k+2}' = 3\left[\frac{T_k}{T_{k+1}}(p_{k+2} - p_{k+1}) + \frac{T_{k+1}}{T_k}(p_{k+1} - p_k)\right].
\tag{13.21}
$$

Applying this equation for each joint ($k = 1, 2, \ldots, n-2$) of the composite curve yields $n - 2$ equations, which is less than the number of unknown derivatives (n). By specifying the derivatives — that is the "speed" — at the endpoints of the composite curve by assigning $p'_0(t_0) = v_{\text{start}}$ and $p'_{n-1}(t_n) = v_{\text{end}}$ (we usually expect objects to be at a standstill before their movement and to stop after accomplishing it, which requires $v_{\text{start}}, v_{\text{end}} = 0$), however, the problem becomes determinant. The linear equation in matrix form is:

$$
\begin{bmatrix}
1 & 0 & \ldots & & & & \\
T_1 & 2(T_0 + T_1) & T_0 & 0 & \ldots & & \\
0 & T_2 & 2(T_1 + T_2) & T_1 & 0 & & \ldots \\
\vdots & & & & & & \\
& & \ldots & 0 & T_{n-1} & 2(T_{n-1} + T_{n-2}) & T_{n-2} \\
& & & \ldots & & 0 & 1
\end{bmatrix}
\begin{bmatrix}
p'_0 \\
p'_1 \\
p'_2 \\
\vdots \\
p'_{n-1} \\
p'_n
\end{bmatrix}
$$

$$
=
\begin{bmatrix}
v_{\text{start}} \\
3[T_0/T_1(p_2 - p_1) + T_1/T_0(p_1 - p_0)] \\
3[T_1/T_2(p_3 - p_2) + T_2/T_1(p_2 - p_1)] \\
\vdots \\
3[T_{n-2}/T_{n-1}(p_n - p_{n-1}) + T_{n-1}/T_{n-2}(p_{n-1} - p_{n-2})] \\
v_{\text{end}}
\end{bmatrix}. \qquad (13.22)
$$

By solving this linear equation, the unknown derivatives $[p'_0, \ldots, p'_n]$ can be determined, which in turn can be substituted into equation 13.20 to define the segments and consequently the composite function $p(t)$.

Cubic spline interpolation produces a C^2 curve from piecewise 3-degree polynomials, thus neither the complexity of the calculations nor the tendency to oscillate increases as the number of knot points increases. The result is a smooth curve exhibiting no variations that are not inherent in the series of knot points, and therefore it can provide realistic animation sequences. This method, however, still has a drawback which appears during the design phase, namely the lack of **local control**. When the animator desires to change a small part of the animation sequence, he will modify a knot point nearest to the timing parameter of the given part. The modification of a single point (either its value or its derivative), however, affects the whole trajectory, since in order to guarantee second order continuity,

the derivatives at the knot points must be recalculated by solving equation 13.22. This may lead to unwanted changes in a part of the trajectory far from the modified knot point, which makes the design process difficult to execute in cases where very fine control is needed. This is why we prefer methods which have this "local control" property, where the modification of a knot point alters only a limited part of the function.

Recall that the representation of cubic polynomials was changed from the set of coefficients to the values and the derivatives at the endpoints when the cubic spline interpolation was introduced. This representation change had a significant advantage in that by forcing two consecutive segments to share two parameters from the four (namely the value and derivative at one endpoint), C^0 and C^1 continuity was automatically guaranteed, and only the continuity of the second derivative had to be taken care of by additional equations. We might ask whether there is another representation of cubic segments which guarantees even C^2 continuity by simply sharing 3 control values from the possible four. There is, namely, the **cubic B-spline**.

The cubic B-spline is a member of a more general family of **k-order B-splines** which are based on a set of k-order (degree $k-1$) blending functions that can be used to define a $p(t)$ function by the linear combination of its knot points $[t_0, p_0; t_1, p_1; \ldots; t_n, p_n]$:

$$p(t) = \sum_{i=0}^{n} p_i \cdot N_{i,k}(t), \qquad k = 2, \ldots n, \qquad (13.23)$$

where the blending functions $N_{i,k}$ are usually defined by the Cox-deBoor recursion formulae:

$$N_{i,1}(t) = \begin{cases} 1 \text{ if } t_i \leq t < t_{i+1} \\ \\ 0 \text{ otherwise} \end{cases} \qquad (13.24)$$

$$N_{i,k}(t) = \frac{(t - t_i)N_{i,k-1}(t)}{t_{i+k-1} - t_i} + \frac{(t_{i+k} - t)N_{i+1,k-1}(t)}{t_{i+k} - t_{i+1}}, \qquad \text{if } k > 1. \quad (13.25)$$

The construction of these blending functions can be interpreted geometrically. At each level of the recursion two subsequent blending functions are taken and they are blended together by linear weighting (see figure 13.5).
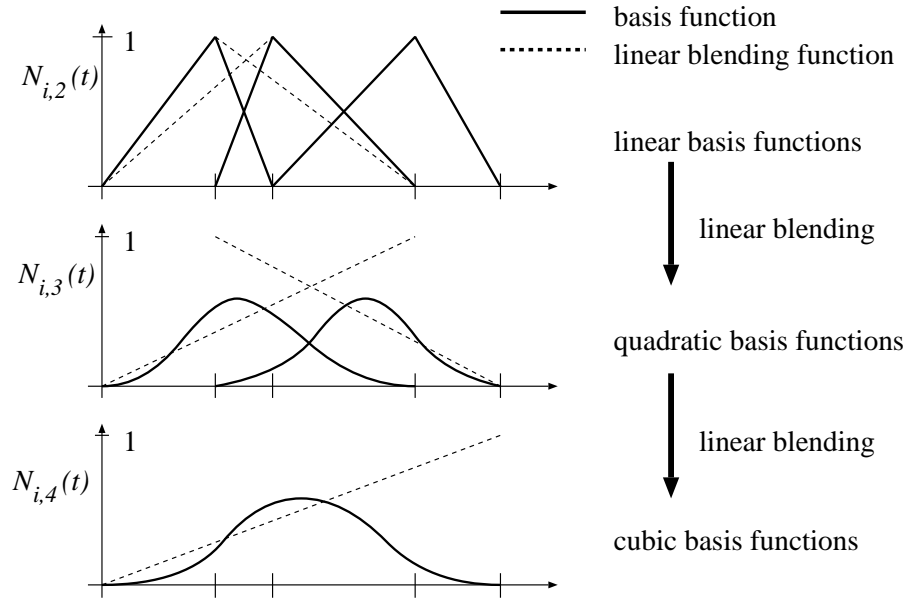
Figure 13.5: Construction of B-spline blending functions

It is obvious from the construction process that $N_{i,k}(t)$ is non-zero only in $[t_i, t_{i+k}]$. Thus a control point p_i can affect the generated

$$p(t) = \sum_{i=0}^{n} p_i \cdot N_{i,k}(t)$$

only in $[t_i, t_{i+k}]$, and therefore the B-spline method has the local control property. The function $p(t)$ is a piecewise polynomial of degree $k-1$, and it can be proven that its derivatives of order $0, 1, \ldots, k-2$ are all continuous at the joints. For animation purposes C^2 continuity is required, thus 4-order (degree 3, that is cubic) B-splines are used.

Examining the cubic B-spline method more carefully, we can see that the interpolation requirement, that is $p(t_i) = p_i$, is not provided, because at t_i more than one blending functions are different from zero. Thus the cubic B-spline offers an **approximation method**. The four blending functions affecting a single point, however, are all positive and their sum is 1, that is, the point is always in the **convex hull** of the 4 nearest control points, and thus the resulting function will follow the polygon of the control points quite reasonably.

The fact that the B-splines offer an approximation method does not mean that they cannot be used for interpolation. If a B-spline which passes through points $[t_0, p_0; t_1, p_1; \ldots; t_n, p_n]$ is needed, then a set of control points $[c_{-1}, c_0, c_1 \ldots c_{n+1}]$ must be found so that:

$$p(t_j) = \sum_{i=-1}^{n+1} c_i \cdot N_{i,k}(t_j) = p_j. \qquad (13.26)$$

This is a linear system of n equations which have $n + 2$ unknown variables. To make the problem determinant the derivatives at the beginning and at the end of the function must be defined. The resulting linear equation can be solved for the unknown control points.

13.5   Interpolation with quaternions

The previous section discussed several trajectory interpolation techniques which determined a time function for each independently controllable motion parameter. These parameters can be used later on to derive the transformation matrix. This two-step method guarantees that the *"inbetweened"* samples are really valid transformations which do not destroy the shape of the animated objects.

Interpolating in the motion parameter space, however, generates new problems which need to be addressed in animation. Suppose, for the sake of simplicity, that an object is to be animated between two different positions and orientations with uniform speed. In parameter space straight line segments are the shortest paths between the two knot points. Unfortunately these line segments do not necessarily correspond to the shortest "natural" path in the space of orientations, only in the space of positions. The core of the problem is the selection of the orientation parameters, that is the roll-pitch-yaw angles, since real objects rotate around a single (time-varying) direction instead of around three superficial coordinate axes, and the dependence of the angle of the single rotation on the roll-pitch-yaw angles is not linear (compare equations 5.26 and 5.30). When rotating by angle α around a given direction in time t, for instance, the linearly interpolated roll-pitch-yaw angles will not necessarily correspond to a rotation by $\lambda \cdot \alpha$ in $\lambda \cdot t$ ($\lambda \in [0..1]$), which inevitably results in uneven and "non-natural"

motion. In order to demonstrate this problem, suppose that an object lo-
cated in [1,0,0] has to be rotated around vector [1,1,1] by 240 degrees and
the motion is defined by three knot points representing rotation by 0, 120
and 240 degrees respectively (figure 13.6). Rotation by 120 degrees moves
the x axis to the z axis and rotation by 240 degrees transforms the x axis to
y axis. These transformations, however, are realized by 90 degree rotations
around the y axis then around the x axis if the roll-pitch-yaw representation
is used. Thus the interpolation in roll-pitch-yaw angles forces the object to
rotate first around the y axis by 90 degrees then around the x axis instead
of rotating continuously around [1,1,1]. This obviously results in uneven
and unrealistic motion even if this effect is decreased by a C^2 interpolation.
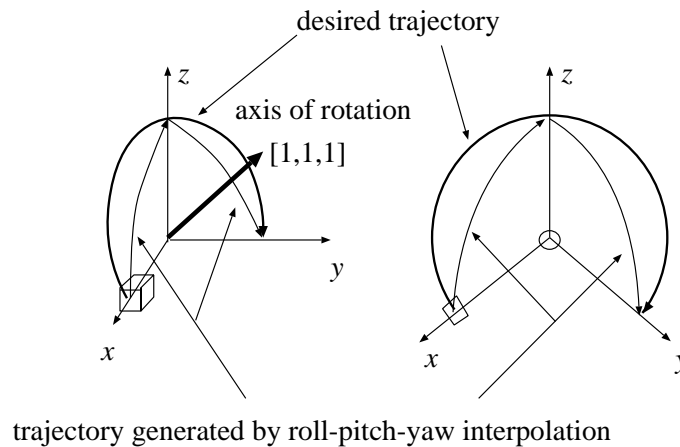
Figure 13.6: Problems of interpolation in roll-pitch-yaw angles

This means that a certain orientation change cannot be inbetweened by
independently interpolating the roll-pitch-yaw angles in cases when these
effects are not tolerable. Rather the axis of the final rotation is required, and
the 2D rotation around this single axis must be interpolated and sampled in
the different frames. Unfortunately, neither the roll-pitch-yaw parameters
nor the transformation matrix supply this required information including
the axis and the angle of rotation. Another representation is needed which
explicitly refers to the axis and the angle of rotation.

In the mid-eighties several publications appeared promoting **quaternions**
as a mathematical tool to describe and handle rotations and orientations in

graphics and robotics [Bra82]. Not only did quaternions solve the problem of natural rotation interpolation, but they also simplified the calculations and out-performed the standard roll-pitch-yaw angle based matrix operations.

Like a matrix, a quaternion q can be regarded as a tool for changing one vector $\vec{u}$ into another $\vec{v}$:

$$\vec{u} \overset{q}{\Longrightarrow} \vec{v}. \tag{13.27}$$

Matrices do this change with a certain element of redundancy, that is, there is an infinite number of matrices which can transform one vector to another given vector. For 3D vectors, the matrices have 9 elements, although 4 real numbers can define this change unambiguously, namely:

1. The change of the length of the vector.

2. The plane of rotation, which can be defined by 2 angles from two given axes.

3. The angle of rotation.

A quaternion q, on the other hand, consists only of the necessary 4 numbers, which are usually partitioned into a pair consisting of a scalar element and a vector of 3 scalars, that is:

$$q = [s, x, y, z] = [s, \vec{w}]. \tag{13.28}$$

Quaternions are four-vectors (this is why they were given this name), and inherit vector operations including addition, scalar multiplication, dot product and norm, but their multiplication is defined specially, in a way somehow similar to the arithmetic of complex numbers, because quaternions can also be interpreted as a generalization of the complex numbers with s as the real part and x, y, z as the imaginary part. Denoting the imaginary axes by $\mathbf{i}, \mathbf{j}$ and $\mathbf{k}$ yields:

$$q = s + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}. \tag{13.29}$$

In fact, Sir Hamilton introduced the quaternions more than a hundred years ago to generalize complex numbers, which can be regarded as *pairs* with special algebraic rules. He failed to find the rules for *triples*, but realized that the generalization is possible for *quadruples* with the rules:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1, \quad \mathbf{ij} = \mathbf{k}, \text{ etc.}$$

To summarize, the definitions of the operations on quaternions are:

$$q_1 + q_2 = [s_1, \vec{w}_1] + [s_2, \vec{w}_2] = [s_1 + s_2, \vec{w}_1 + \vec{w}_2],$$

$$\lambda q = \lambda[s, \vec{w}] = [\lambda s, \lambda \vec{w}],$$

$$q_1 \cdot q_2 = [s_1, \vec{w}_1] \cdot [s_2, \vec{w}_2] = [s_1 s_2 - \vec{w}_1 \cdot \vec{w}_2, s_1 \vec{w}_2 + s_2 \vec{w}_1 + \vec{w}_1 \times \vec{w}_2],$$

$$\langle q_1, q_2 \rangle = \langle [s_1, x_1, y_1, z_1], [s_2, x_2, y_2, z_2] \rangle = s_1 s_2 + x_1 x_2 + y_1 y_2 + z_1 z_2,$$

$$||q|| = ||[s, x, y, z]|| = \sqrt{\langle q, q \rangle} = \sqrt{s^2 + x^2 + y^2 + z^2}.$$

$$(13.30)$$

Quaternion multiplication and addition satisfies the distributive law. Addition is commutative and associative. Multiplication is associative but is not commutative. It can easily be shown that the multiplicative identity is $I = [1, \vec{0}]$. With respect to quaternion multiplication the inverse quaternion is:

$$q^{-1} = \frac{[s, -\vec{w}]}{||q||^2} \tag{13.31}$$

since

$$[s, \vec{w}] \cdot [s, -\vec{w}] = [s^2 + |\vec{w}|^2, \vec{0}] = ||q||^2 \cdot [1, \vec{0}]. \tag{13.32}$$

As for matrices, the inverse reverses the order of multiplication, that is:

$$(q_2 \cdot q_1)^{-1} = q_1^{-1} \cdot q_2^{-1}. \tag{13.33}$$

Our original goal, the rotation of 3D vectors using quaternions, can be achieved relying on quaternion multiplication by having extended the 3D vector by an $s = 0$ fourth parameter to make it, too, a quaternion:

$$\vec{u} \stackrel{q}{\Longrightarrow} \vec{v} : \qquad [0, \vec{v}] = q \cdot [0, \vec{u}] \cdot q^{-1} =$$

$$\frac{[0, s^2 \vec{u} + 2s(\vec{w} \times \vec{u}) + (\vec{w} \cdot \vec{u})\vec{w} + \vec{w} \times (\vec{w} \times \vec{u})]}{||q||^2}. \tag{13.34}$$

Note that a scaling in quaternion $q = [s, \vec{w}]$ makes no difference to the resulting vector v, since scaling of $[s, \vec{w}]$ and $[s, -\vec{w}]$ in q^{-1} is compensated for by the attenuation of $||q||^2$. Thus, without the loss of generality, we assume that q is a unit quaternion, that is

$$||q||^2 = s^2 + |\vec{w}|^2 = 1 \tag{13.35}$$

For unit quaternions, equation 13.34 can also be written as:

$$[0, \vec{v}] = q \cdot [0, \vec{u}] \cdot q^{-1} = [0, \vec{u} + 2s(\vec{w} \times \vec{u}) + 2\vec{w} \times (\vec{w} \times \vec{u})] \qquad (13.36)$$

since

$$s^2\vec{u} = \vec{u} - |\vec{w}|^2\vec{u} \quad \text{and} \quad (\vec{w} \cdot \vec{u})\vec{w} - |\vec{w}|^2\vec{u} = \vec{w} \times (\vec{w} \times \vec{u}). \qquad (13.37)$$

In order to examine the effects of the above defined transformation, vector $\vec{u}$ is first supposed to be perpendicular to vector $\vec{w}$, then the parallel case will be analyzed.

If vector $\vec{u}$ is perpendicular to quaternion element $\vec{w}$, then for unit quaternions equation 13.36 yields:

$$q \cdot [0, \vec{u}] \cdot q^{-1} = [0, \vec{u}(1 - 2|\vec{w}|^2) + 2s(\vec{w} \times \vec{u})] = [0, \vec{v}]. \qquad (13.38)$$
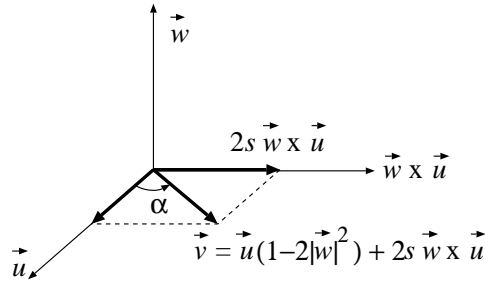
Figure 13.7: Geometry of quaternion rotation for the perpendicular case

That is, $\vec{v}$ is a linear combination of perpendicular vectors $\vec{u}$ and $\vec{w} \times \vec{u}$ (figure 13.7), it thus lies in the plane of $\vec{u}$ and $\vec{w} \times \vec{u}$, and its length is:

$$|\vec{v}| = |\vec{u}|\sqrt{(1 - 2|\vec{w}|^2)^2 + (2s|\vec{w}|)^2} = |\vec{u}|\sqrt{(1 + 4|\vec{w}|^2(s^2 + |\vec{w}|^2 - 1)} = |\vec{u}|.$$
$$(13.39)$$

Since $\vec{w}$ is perpendicular to the plane of $\vec{u}$ and the resulting vector $\vec{v}$, and the transformation does not alter the length of the vector, vector $\vec{v}$ is, in fact, a rotation of $\vec{u}$ around $\vec{w}$. The cosine of the rotation angle (α) can be expressed by the dot product of $\vec{u}$ and $\vec{v}$, that is:

$$\cos\alpha = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}| \cdot |\vec{v}|} = \frac{(\vec{u} \cdot \vec{u})(1 - 2|\vec{w}|^2) + 2s\vec{u} \cdot (\vec{w} \times \vec{u})}{|\vec{u}|^2} = 1 - 2|\vec{w}|^2. \quad (13.40)$$

If vector $\vec{u}$ is parallel to quaternion element $\vec{w}$, then for unit quaternions equation 13.36 yields:

$$[0, \vec{v}] = q \cdot [0, \vec{u}] \cdot q^{-1} = [0, \vec{u}]. \qquad (13.41)$$

Thus the parallel vectors are not affected by quaternion multiplication as rotation does not alter the axis parallel vectors.

General vectors can be broken down into a parallel and a perpendicular component with respect to $\vec{w}$ because of the distributive property. As has been demonstrated, the quaternion transformation rotates the perpendicular component by an angle that satisfies $\cos \alpha = 1 - 2|\vec{w}|^2$ and the parallel component is unaffected, thus the transformed components will define the rotated version of the original vector $\vec{u}$ by angle α around the vector part of the quaternion.

Let us apply this concept in the reverse direction and determine the rotating quaternion for a required rotation axis $\vec{d}$ and angle α. We concluded that the quaternion transformation rotates the vector around its vector part, thus a unit quaternion rotating around unit vector $\vec{d}$ has the following form:

$$q = [s, r \cdot \vec{d}], \qquad s^2 + r^2 = 1 \qquad (13.42)$$

Parameters s and r have to be selected according to the requirement that quaternion q must rotate by angle α. Using equations 13.40 and 13.42, we get:

$$\cos \alpha = 1 - 2r^2, \qquad s = \sqrt{1 - r^2}. \qquad (13.43)$$

Expressing parameters s and r, then quaternion q that represents a rotation by angle α around a unit vector $\vec{d}$, we get:

$$q = [\cos \frac{\alpha}{2}, \sin \frac{\alpha}{2} \cdot \vec{d}]. \qquad (13.44)$$

The special case when $\sin \alpha/2 = 0$, that is $\alpha = 2k\pi$ and $q = [\pm 1, \vec{0}]$, poses no problem, since a rotation of an even multiple of π does not affect the object, and the axis is irrelevant.

Composition of rotations is the "concatenation" of quaternions as in matrix representation since:

$$q_2 \cdot (q_1 \cdot [0, \vec{u}] \cdot q_1^{-1}) \cdot q_2^{-1} = (q_2 \cdot q_1) \cdot [0, \vec{u}] \cdot (q_2 \cdot q_1)^{-1}. \qquad (13.45)$$

Let us focus on the interpolation of orientations between two knot points in the framework of quaternions. Suppose that the orientations are described in the two knot points by quaternions q_1 and q_2 respectively. For the sake of simplicity, we suppose first that q_1 and q_2 represent rotations around the same unit axis $\vec{d}$, that is:

$$q_1 = [\cos \frac{\alpha_1}{2}, \sin \frac{\alpha_1}{2} \cdot \vec{d}], \qquad q_2 = [\cos \frac{\alpha_2}{2}, \sin \frac{\alpha_2}{2} \cdot \vec{d}]. \tag{13.46}$$

Calculating the dot product of q_1 and q_2,

$$\langle q_1, q_2 \rangle = \cos \frac{\alpha_1}{2} \cdot \cos \frac{\alpha_2}{2} + \sin \frac{\alpha_1}{2} \cdot \sin \frac{\alpha_2}{2} = \cos \frac{\alpha_2 - \alpha_1}{2},$$

we come to the interesting conclusion that the angle of rotation between the two orientations represented by the quaternions is, in fact, half of the angle between the two quaternions in 4D space.
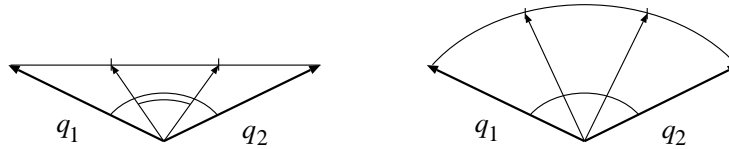
Figure 13.8: Linear versus spherical interpolation of orientations

Our ultimate objective is to move an object from an orientation represented by q_1 to a new orientation of q_2 by an even and uniform motion. If linear interpolation is used to generate the path of orientations between q_1 and q_2, then the angles of the subsequent quaternions will not be constant, as is demonstrated in figure 13.8. Thus the speed of the rotation will not be uniform, and the motion will give an effect of acceleration followed by deceleration, which is usually undesirable.

Instead of linear interpolation, a non-linear interpolation must be found that guarantees the constant angle between the subsequent interpolated quaternions. Spherical interpolation obviously meets this requirement, where the interpolated quaternions are selected uniformly from the arc between q_1 and q_2. If q_1 and q_2 are unit quaternions, then all the interpolated quaternions will also be of unit length. Unit-size quaternions can be regarded

as unit-size four-vectors which correspond to a 4D unit-radius sphere. An appropriate interpolation method must generate the great arc between q_1 and q_2, and as can easily be shown, this great arc has the following form:

$$q(t) = \frac{\sin(1-t)\theta}{\sin\theta} \cdot q_1 + \frac{\sin t\theta}{\sin\theta} \cdot q_2, \qquad (13.47)$$

where $\cos\theta = \langle q_1, q_2 \rangle$ (figure 13.9).
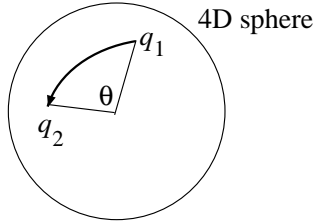
Figure 13.9: Interpolation of unit quaternions on a 4D unit sphere

In order to demonstrate that this really results in a uniform interpolation, the following equations must be proven for $q(t)$:

$$||q(t)|| = 1, \quad \langle q_1, q(t) \rangle = \cos(t\theta), \quad \langle q_2, q(t) \rangle = \cos((1-t)\theta). \qquad (13.48)$$

That is, the interpolant is really on the surface of the sphere, and the angle of rotation is a linear function of the time t.

Let us first prove the second assertion (the third can be proven similarly):

$$\langle q_1, q(t) \rangle = \frac{\sin(1-t)\theta}{\sin\theta} + \frac{\sin t\theta}{\sin\theta} \cdot \cos\theta =$$

$$\frac{\sin\theta \cdot \cos t\theta}{\sin\theta} - \frac{\sin t\theta \cdot \cos\theta}{\sin\theta} + \frac{\sin t\theta}{\sin\theta} \cdot \cos\theta = \cos(t\theta). \qquad (13.49)$$

Concerning the norm of the interpolant, we can use the definition of the norm and the previous results, thus:

$$||q(t)||^2 = \langle q(t), q(t) \rangle = \langle \frac{\sin(1-t)\theta}{\sin\theta} \cdot q_1 + \frac{\sin t\theta}{\sin\theta} \cdot q_2, q(t) \rangle =$$

$$\frac{\sin(1-t)\theta}{\sin\theta} \cdot \cos(t\theta) + \frac{\sin t\theta}{\sin\theta} \cdot \cos((1-t)\theta) = \frac{\sin((1-t)\theta + t\theta)}{\sin\theta} = 1 \quad (13.50)$$

If there is a series of consecutive quaternions $q_1, q_2, \ldots, q_n$ to follow during the animation, this interpolation can be executed in a similar way to that discussed in the previous section. The blending function approach can be used, but here the constraints are slightly modified. Supposing $b_1(t), b_2(t), \ldots, b_n(t)$ are blending functions, for any t, they must satisfy that:

$$||b_1(t)q_1 + b_2(t)q_2 + \ldots + b_n(t)q_n|| = 1 \qquad (13.51)$$

which means that the curve must lie on the sphere. This is certainly more difficult than generating a curve in the plane of the control points, which was done in the previous section. Selecting $b_i(t)$s as piecewise curves defined by equation 13.47 would solve this problem, but the resulting curve would not be of C^1 and C^2 type.

Shoemake [Sho85] proposed a successive linear blending technique on the surface of the sphere to enforce the continuous derivatives. Suppose that a curve segment adjacent to $q_1, q_2, \ldots q_n$ is to be constructed. In the first phase, piecewise curves are generated between q_1 and q_2, q_2 and q_3, etc.:

$$q^{(1)}(t_1) = \frac{\sin(1 - t_1)\theta_1}{\sin \theta_1}q_1 + \frac{\sin t_1 \theta_1}{\sin \theta_1}q_2,$$

$$q^{(2)}(t_2) = \frac{\sin(1 - t_2)\theta_2}{\sin \theta_2}q_2 + \frac{\sin t_2 \theta_2}{\sin \theta_2}q_3,$$

$$\vdots$$

$$q^{(n-1)}(t_{n-1}) = \frac{\sin(1 - t_{n-1})\theta_{n-1}}{\sin \theta_{n-1}}q_{n-1} + \frac{\sin t_{n-1} \theta_{n-1}}{\sin \theta_{n-1}}q_n. \qquad (13.52)$$

In the second phase these piecewise segments are blended to provide a higher order continuity at the joints. Let us mirror q_{i-1} with respect to q_i on the sphere generating q_{i-1}^*, and determine the point a_i that bisects the great arc between q_{i+1} and q_{i-1}^* (see figure 13.10). Let us form another great arc by mirroring a_i with respect to q_i generating a_i^* as the other endpoint. Having done this, a C^2 approximating path $g(t)$ has been produced from the neighborhood of q_{i-1} (since $q_{i-1} \approx a_i^*$) through q_i to the neighborhood of q_{i+1} (since $q_{i+1} \approx a_i$). This great arc is subdivided into two segments producing $g^{(i-1)}(t)$ between a_i^* and q_i and $g^{(i)}(t)$ between q_i and a_i respectively.

In order to guarantee that the final curve goes through q_{i-1} and q_{i+1} without losing its smoothness, a linear blending is applied between the

piecewise curves $q^{(i-1)}(t)$, $q^{(i)}(t)$ and the new approximation arcs $g^{(i-1)}(t)$, $g^{i}(t)$ in such a way that the blending gives weight 1 to $q^{(i-1)}(t)$ at $t_{i-1} = 0$, to $q^{(i+1)}(t)$ at $t_i = 1$ and to the approximation arcs $g^{(i-1)}$ and $g^{(i)}$ at $t_{i-1} = 1$ and $t_i = 0$ respectively, that is:

$$\hat{q}^{(i-1)}(t_{i-1}) = (1 - t_{i-1}) \cdot q^{(i-1)}(t_{i-1}) + t_{i-1} \cdot g^{(i-1)}(t_{i-1})$$
$$\hat{q}^{(i)}(t_i) = (1 - t_i) \cdot g^{(i)}(t_i) + t_i \cdot q^{(i)}(t_i). \tag{13.53}$$

This method requires uniform timing between the successive knot points. By applying a linear transformation in the time domain, however, any kind of timing can be specified.
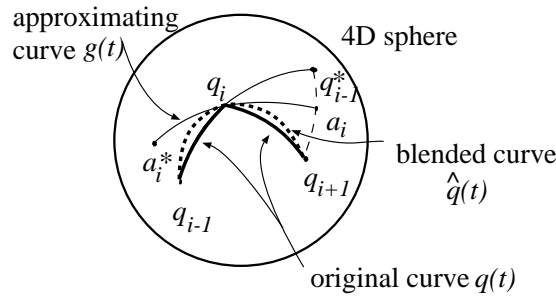
Figure 13.10: Shoemake's algorithm for interpolation of unit quaternions on a 4D unit sphere

Once the corresponding quaternion of the interpolated orientation is determined for a given time parameter t, it can be used for rotating the objects in the model. Comparing the number of instructions needed for (spherical) quaternion interpolation and rotation of the objects by quaternion multiplication, we can see that the method of quaternions not only provides more realistic motion but is slightly more effective computationally.

However, the traditional transformation method based on matrices can be combined with this new approach using quaternions. Using the interpolated quaternion a corresponding transformation matrix can be set up. More precisely this is the upper-left minor matrix of the transformation matrix, which is responsible for the rotation, and the last row is the position vector which is interpolated by the usual techniques. In order to identify the transformation matrix from a quaternion, the way the basis vectors are

transformed when multiplied by the quaternion must be examined. By applying unit quaternion $q = [s, x, y, z]$ to the first, second and third standard basis vectors $[1,0,0]$, $[0,1,0]$ and $[0,0,1]$, the first, second and the third rows of the matrix can be determined, thus:

$$\mathbf{A}_{3\times 3} = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy + 2sz & 2xz - 2sy \\ 2xy - 2sz & 1 - 2x^2 - 2z^2 & 2yz + 2sx \\ 2xz + 2sy & 2yz - 2sx & 1 - 2x^2 - 2y^2 \end{bmatrix}. \qquad (13.54)$$

During the interactive design phase of the animation sequence, we may need the inverse conversion which generates the quaternion from the (orthonormal) upper-left part of the transformation matrix or from the roll-pitch-yaw angles. Expressing $[s, x, y, z]$ from equation 13.54 we get:

$$s = \sqrt{1 - (x^2 + y^2 + z^2)} = \frac{1}{2}\sqrt{a_{11} + a_{22} + a_{33} + 1},$$

$$x = \frac{a_{23} - a_{32}}{4s}, \quad y = \frac{a_{31} - a_{13}}{4s}, \quad z = \frac{a_{12} - a_{21}}{4s}. \qquad (13.55)$$

The roll-pitch-yaw (α, β, γ) description can also be easily transformed into a quaternion if the quaternions corresponding to the elementary rotations are combined:

$$q(\alpha, \beta, \gamma) = [\cos\frac{\alpha}{2}, (0, 0, \sin\frac{\alpha}{2})] \cdot [\cos\frac{\beta}{2}, (0, \sin\frac{\beta}{2}, 0)] \cdot [\cos\frac{\gamma}{2}, (\sin\frac{\gamma}{2}, 0, 0)].$$
$$(13.56)$$

13.6 Hierarchical motion

So far we have been discussing the animation of individual rigid objects whose paths could be defined separately taking just several collision constraints into consideration. To avoid unexpected collisions in these cases, the animation sequence should be reviewed and the definition of the keyframes must be altered iteratively until the animation sequence is satisfactory.

Real objects usually consist of several **linked segments**, as for example a human body is composed of a trunk, a head, two arms and two legs. The arms can in turn be broken down into an upper arm, a lower arm, hand, fingers etc. A car, on the other hand, is an assembly of its body and the four wheels (figure 13.11). The segments of a composed object (an

assembly) do not move independently, because they are linked together by joints which restrict the relative motion of linked segments. Revolute joints, such as human joints and the coupling between the wheel and the body of the car, allow for specific rotations about a fixed common point of the two linked segments. Prismatic joints, common in robots and in machines [Lan91], however, allow the parts to translate in a given direction. When these assembly structures are animated, the constraints generated by the features of the links must be satisfied in every single frame of the animation sequence. Unfortunately, it is not enough to meet this requirement in the keyframes only and animate the segments separately. A running human body, for instance, can result in frames when the trunk, legs, and the arms are separated even if they are properly connected in the keyframes. In order to avoid these annoying effects, the constraints and relationships of the various segments must continuously be taken into consideration during the interpolation, not just in the knot points. This can be achieved if the segments are not animated separately but their relative motion is specified.
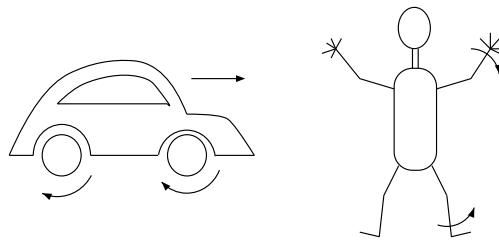
Figure 13.11: Examples of multi-segment objects

Recall that the motion of an individual object is defined by a time-varying modeling transformation matrix which places the object in the common world coordinate system. If the relative motion of object i must be defined with respect to object j, then the relative modeling transformation $\mathbf{T_{ij}}$ of object i must place it in the local coordinate system of object j. Since object j is fixed in its own modeling coordinate system, $\mathbf{T_{ij}}$ will determine the relative position and orientation of object i with respect to object j. A point $\vec{r}_i$ in object i's coordinate system will be transformed to point:

$$[\vec{r}_j, 1] = [\vec{r}_i, 1] \cdot \mathbf{T_{ij}} \qquad \Longrightarrow \qquad \vec{r}_j = \vec{r}_i \cdot \mathbf{A_{ij}} + \vec{p}_{ij} \qquad (13.57)$$

in the local modeling system of object j if $\mathbf{A_{ij}}$ and $\vec{p}_{ij}$ are the orientation matrix and translation vector of matrix $\mathbf{T_{ij}}$ respectively. While animating this object, matrix $\mathbf{T_{ij}}$ is a function of time. If only orientation matrix $\mathbf{A_{ij}}$ varies with time, the relative position of object i and object j will be fixed, that is, the two objects will be linked together by a revolute joint at point $\vec{p}_{ij}$ of object j and at the center of its own local coordinate system of object i. Similarly, if the orientation matrix is constant in time, but the position vector is not, then a prismatic joint is simulated which allows object i to move anywhere but keeps its orientation constant with respect to object j.

Transformation $\mathbf{T_{ij}}$ places object i in the local modeling coordinate system of object j. Thus, the world coordinate points of object i can be generated if another transformation — object j's modeling transformation $\mathbf{T_j}$ which maps the local modeling space of object j onto world space — is applied:

$$[\vec{r}_w, 1] = [\vec{r}_j, 1] \cdot \mathbf{T_j} = [\vec{r}_i, 1] \cdot \mathbf{T_{ij}} \cdot \mathbf{T_j}. \tag{13.58}$$

In this way, whenever object j is moved, object i will follow it with a given relative orientation and position since object j's local modeling transformation will affect object i as well. Therefore, object j is usually called the **parent segment** of object i and object i is called the **child segment** of object j. A child segment can also be a parent of other segments. In a simulated human body, for instance, the upper arm is the child of the trunk, in turn is the parent of the lower arm (figure 13.12). The lower arm has a child, the hand, which is in turn the parent of the fingers. The parent-child relationships form a *hierarchy of segments* which is responsible for determining the types of motion the assembly structure can accomplish. This hierarchy usually corresponds to a *tree-structure* where a child has only one parent, as in the examples of the human body or the car. The motion of an assembly having a tree-like hierarchy can be controlled by defining the modeling transformation of the complete structure and the relative modeling transformation for every single parent-child pair (joints in the assembly). In order to set these transformations, the normal interactive techniques can be used. First we move the complete human body (including the trunk, arms, legs etc.), then we arrange the arms (including the lower arms, hand etc.) and legs, then the lower arms, hands, fingers etc. by interactive manipulation. In the animation design program, this interactive manipulation updates the modeling transformation of the body first, then the relative modeling trans-
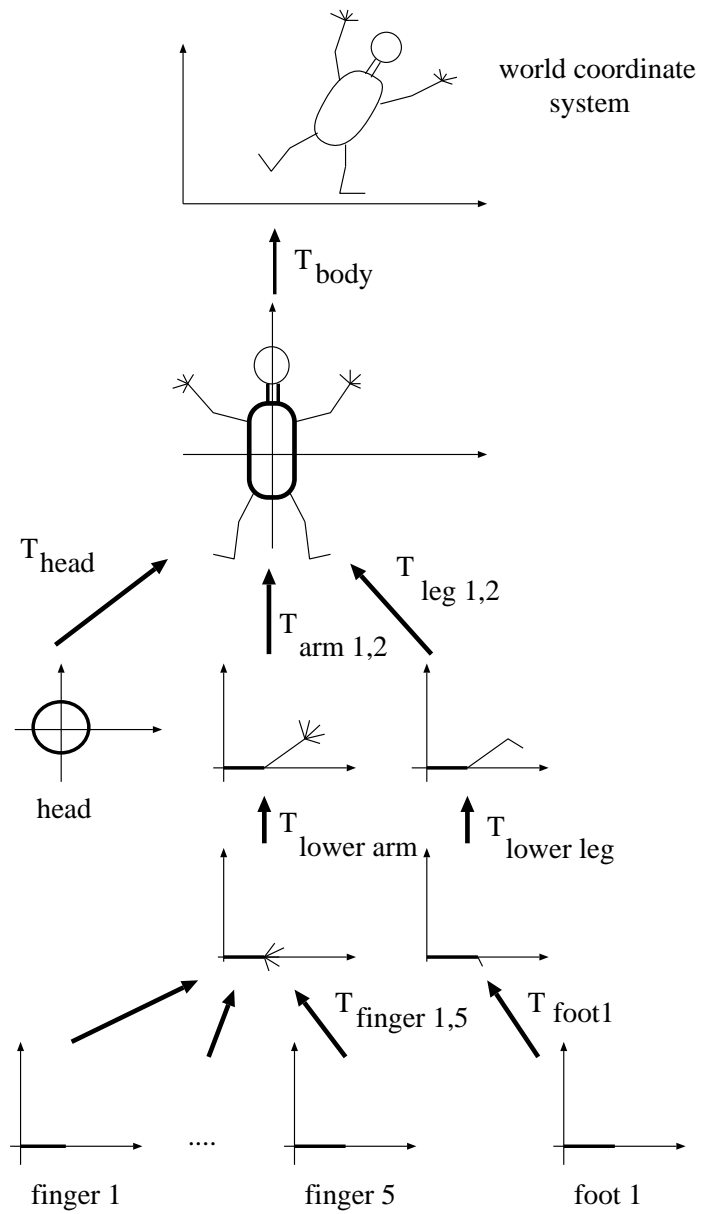
Figure 13.12: Transformation tree of the human body

formation of the arms and legs, then the relative transformation of the lower arms etc. Thus, in each keyframe, the modeling transformation in the joints of hierarchy can be defined. During interpolation, these transformations are interpolated independently and meeting the requirements of the individual joints (in a revolute joint the relative position is constant), but the overall transformation of a segment is generated by the concatenation of the relative transformations of its ancestors and the modeling transformation of the complete assembly structure. This will guarantee that the constraints imposed by the joints in the assembly will be satisfied.
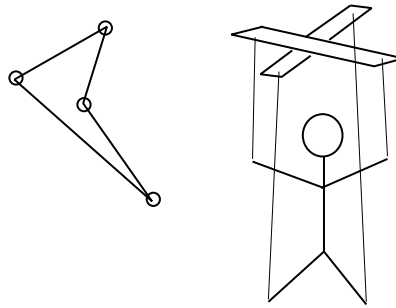
Figure 13.13: Assemblies having non tree-like segment structure

13.6.1   Constraint-based systems

In tree-like assemblies independent interpolation is made possible by the assumption that each joint enjoys independent degree(s) of freedom in its motion. Unfortunately, this assumption is not always correct, and this can cause, for example, the leg to go through the trunk which is certainly not "realistic". Most of these collision problems can be resolved by reviewing the animation sequence that was generated without taking care of the collisions and the interdependence of various joints, and modifying the keyframes until a satisfactory result is generated. This try-and-check method may still work for problems where there are several objects in the scene and their collision must be avoided, but this can be very tiresome, so we would prefer methods which resolve these collision and interdependence problems automatically. The application of these automatic constraint resolution methods is essential

for non tree-like assemblies (figure 13.13), where the degree of freedom is less than the individually controllable parameters of the joints, because the independent interpolation of a subset of joint parameters may cause other joints to fall apart even if all requirements are met in the keyframes.

Such an automatic constraint resolution algorithm basically does the same as the user who interactively tries to modify the definition of the sequence and checks whether or not the result satisfied the constraints. The algorithm is controlled by an *error* function which is non-zero if a constraint is not satisfied and usually increases as we move away from the allowed arrangements. The motion algorithm tries to minimize this function by interpolating a C^2 function for each controllable parameter, calculating the maximum of this error function along the path and then modifying the knot points of the C^2 function around the parameters where the error value is large. Whenever a knot point is modified, the trajectory is evaluated again and a check is made to see the error value has decreased or not. If it has decreased, then the previous modification is repeated; if it has increased, the previous modification is inverted. The new parameter knot point should be randomly perturbed to avoid infinite oscillations and to reduce the probability of reaching a local minimum of the error function. The algorithm keeps repeating this step until either it can generate zero error or it decides that no convergence can be achieved possibly because of overconstraining the system. This method is also called the **relaxation technique**.

When animating complex structures, such as the model of the human body, producing the effect of realistic motion can be extremely difficult and can require a lot of expertise and experience of traditional cartoon designers. The C^2 interpolation of the parameters is a necessary but not a sufficient requirement for this. Generally, the real behavior and the internal structure of the simulated objects must be understood in order to imitate their motion. Fortunately, the most important rule governing the motion of animals and humans is very simple: *Living objects always try to minimize the energy needed for a given change of position and orientation and the motion must satisfy the geometric constraints of the body and the dynamic constraints of the muscles.* Thus, when the motion parameters are interpolated between the keyframe positions, the force needed in the different joints as well as the potential and kinetic energy must be calculated. This seems simple, but the actual calculation can be very complex. Fortunately, the same problems have arisen in the control of robots, and therefore the solution methods de-

veloped for robotics can also be used here [Lan91]. The previous relaxation technique must be extended to find not only a trajectory where the error including the geometric and dynamic constraints is zero, but one where the energy generated by the "muscles" in the joints is minimal.

Finally it must be mentioned that an important field of animation, called the **scientific visualization**, focuses on the behavior of systems that are described by a set of physical laws. The objective is to find an arrangement or movement that satisfies these laws.

13.7 Double buffering

Animation means the fast generation of images shown one after the other on the computer screen. If the display of these static images takes a very short time, the human eye is unable to identify them as separate pictures, but rather interprets them as a continuously changing sequence. This phenomenon is well known and is exploited in the motion picture industry.
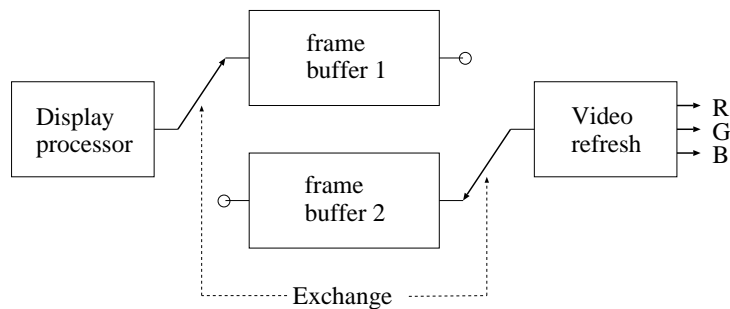
Figure 13.14: Double buffer animation systems

When an image is generated on the computer screen, it usually evolves gradually, depending on the actual visibility algorithm. Painter's algorithm, for example, draws the polygons in order of distance from the camera; thus even those polygons that turn out to be invisible later on will be seen on the screen for a very short time during the image generation. The z-buffer algorithm, on the other hand, draws a polygon point if the previously displayed polygons do not hide it, which can also cause the temporary display of invisible polygons. The evolution of the images, even if it takes a very

short time, may cause noticeable and objectionable effects which need to be eliminated. We must prevent the observer from seeing the generation process of the images and present to him the final result only. This problem had to be solved in traditional motion pictures as well. The usual way of doing it is via the application of two frame buffers, which leads to a method of **double-buffer animation** (figure 13.14). In each frame of the animation sequence, the content of one of the frame buffers is displayed, while the other is being filled up by the image generation algorithm. Once the image is complete, the roles of the two frame buffers are exchanged. Since it takes practically no time — being only a switch of two multiplexers during the vertical retrace — only complete images are displayed on the screen.

13.8 Temporal aliasing

As has been mentioned, animation is a fast display of static image sequences providing the illusion of continuous motion. This means that the motion must be sampled in discrete time instances and then the "continuous" motion produced by showing these static images until the next sampling point. Thus, sampling artifacts, called **temporal aliasing**, can occur if the sampling frequency and the frequency range of the motion do not satisfy the sampling theorem. Well-known examples of temporal aliasing are backward rotating wheels and the jerky motion which can be seen in old movies. These kinds of temporal aliasing phenomena are usually called **strobing**. Since the core of the problem is the same as spatial aliasing due to the finite resolution raster grid, similar approaches can be applied to solve it, including either post-filtering with supersampling, which generates several images in each frame time and produces the final one as their average, or pre-filtering, which solves the visibility and shading problems as a function of time and calculates the convolution of the time-varying image with an appropriate filter function. The filtering process will produce **motion blur** for fast moving objects just as moving objects cause blur on normal films because of finite exposure time. Since visibility and shading algorithms have been developed to deal with static object spaces and images, and most of them are not appropriate for a generalization to take time-varying phenomena into account, temporal anti-aliasing methods usually use a combination of post-filtering and supersampling. (An exceptional case is a kind of ray

tracing which allows for some degree of dynamic generalization as proposed by Cook [CPC84] creating a method called *distributed ray tracing*.)

Let ΔT be the interval during which the images, called **subframes**, are averaged. This time is analogous to the exposure time when the shutter is open in a normal camera. If n number of subframes are generated and box filtering is used, then the averaged color at some point of the image is:

$$I = \frac{1}{n} \sum_{i=0}^{n-1} I(t_0 + \frac{i \cdot \Delta T}{n}). \tag{13.59}$$

The averaging calculation can be executed in the frame buffer. Before writing a pixel value into the frame buffer, its red, green and blue components must be divided by n, and the actual *pixel operation* must be set to "arithmetic addition". The number of samples, n, must be determined to meet (at least approximately) the requirements of the sampling theorem, taking the temporal frequencies of the motion into consideration. Large n values, however, are disadvantageous because temporal supersampling increases the generation time of the animation sequence considerably. Fortunately, acceptable results can be generated with relatively small n if this method is combined with **stochastic sampling** (see section 11.4), that is, if the sample times of the subframes are selected randomly rather than uniformly in the frame interval. Stochastic sampling will transform temporal aliasing into noise appearing as motion blur. Let δ be a random variable distributed in [0,1] to perturb the uniform sample locations. The modified equation to calculate the averaged color is:

$$I = \frac{1}{n} \sum_{i=0}^{n-1} I(t_0 + \frac{(i + \delta) \cdot \Delta T}{n}). \tag{13.60}$$

Temporal filtering can be combined with spatial filtering used to eliminate the "jaggies" [SR92]. Now an image (frame) is averaged from n static images. If these static images are rendered assuming a slightly shifting pixel grid, then the averaging will effectively cause the static parts of the image to be box filtered. The shift of the pixel grid must be evenly distributed in $[(0,0) \ldots (1,1)]$ assuming pixel coordinates. This can be achieved by the proper control of the real to integer conversion during image generation. Recall that we used the Trunc function to produce this, having added 0.5 to the values in the initialization phase. By modifying this 0.5 value in the range of [0,1], the shift of the pixel grid can be simulated.

Bibliography

[AB87]   S. Abhyankar and C. Bajaj. Automatic parametrization of rational
           curves and surfaces II: conics and conicoids. *Computer News*, 25(3),
           1987.

[Ae91]   James Arvo (editor). *Graphics Gems II.* Academic Press, San Diego,
           CA., 1991.

[AK87]   James Arvo and David Kirk. Fast ray tracing by ray classification. In
           Proceedings of SIGGRAPH '87, Computer Graphics, pages 55–64,
           1987.

[ANW67]   J. Ahlberg, E. Nilson, and J. Walsh. *The Theory of Splines and
           their Applications.* Academic Press, 1967.

[Arv91a]   James Arvo. Linear-time voxel walking for octrees. *Ray Trac-
           ing News*, 1(2), 1991. available under anonymous ftp from
           weedeater.math.yale.edu.

[Arv91b]   James Arvo. Random rotation matrices. In James Arvo, editor,
           Graphics Gems II, pages 355–356. Academic Press, Boston, 1991.

[Ath83]   Peter R. Atherton. A scan-line hidden surface removal for construc-
           tive solid geometry. In *Proceedings of SIGGRAPH '83, Computer
           Graphics*, pages 73–82, 1983.

[AW87]   John Amanatides and Andrew Woo. A fast voxel traversal algorithm
           for ray tracing. In *Proceedings of Eurographics '87*, pages 3–10, 1987.

[AWG78]   P. Atherton, K. Weiler, and D. Greenberg. Polygon shadow gener-
           ation. *Computer Graphics*, 12(3):275–281, 1978.

[Bal62]   A.V. Balakrishnan. On the problem of time jitter in sampling. *IRE
           Trans. Inf. Theory*, Apr:226–236, 1962.

[Bar86]   Alan H. Barr. Ray tracing deformed surfaces. In *Proceedings of SIGGRAPH '86, Computer Graphics*, pages 287–296, 1986.

[Bau72]   B.G. Baumgart. Winged-edge polyhedron representation. Technical Report STAN-CS-320, Computer Science Department, Stanford University, Palo Alto, CA, 1972.

[BBB87]   R. Bartels, J. Beatty, and B. Barsky. *An Introduction on Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufmann, Los Altos, CA, 1987.

[BC87]   Ezekiel Bahar and Swapan Chakrabarti. Full wave theory applied to computer-aided graphics for 3d objects. *IEEE Computer Graphics and Applications*, 7(7):11–23, 1987.

[BDH+89]   G. R. Beacon, S. E. Dodsworth, S. E. Howe, R. G. Oliver, and A. Saia. Boundary evaluation using inner and outer sets: The isos method. *IEEE Computer Graphics and Applications*, 9(March):39–51, 1989.

[Bez72]   P. Bezier. *Numerical Control: Mathematics and Applications*. Wiley, Chichester, 1972.

[Bez74]   P. Bezier. *Mathematical and Practical Possibilities of UNISURF*. Academic Press, New York, 1974.

[BG86]   Carlo Braccini and Marino Giuseppe. Fast geometrical manipulations of digital images. *Computer Graphics and Image Processing*, 13:127–141, 1986.

[BG89]   Peter Burger and Duncan Gillies. *Interactive Computer Graphics: Functional, Procedural and Device-Level Methods*. Addison-Wesley, Wokingham, England, 1989.

[Bia90]   Buming Bian. *Accurate Simulation of Scene Luminances*. PhD thesis, Worcester Polytechnic Institute, Worcester, Mass., 1990.

[Bia92]   Buming Bian. Hemispherical projection of a triangle. In David Kirk, editor, *Graphics Gems III*, pages 314–317. Academic Press, Boston, 1992.

[BKP92a]   Jeffrey C. Beran-Koehn and Mark J. Pavicic. A cubic tetrahedra adaption of the hemicube algorithm. In David Kirk, editor, *Graphics Gems II*, pages 324–328. Academic Press, Boston, 1992.

[BKP92b]   Jeffrey C. Beran-Koehn and Mark J. Pavicic. Delta form-factor cal-
            culation for the cubic tetrahedral algorithm. In David Kirk, editor,
            Graphics Gems III, pages 324–328. Academic Press, Boston, 1992.

[Bli77]   James F. Blinn. Models of light reflection for computer synthe-
            sized pictures. In *SIGGRAPH 1977 Proceedings, Computer Graph-
            ics*, pages 192–198, 1977.

[Bli78]   James F. Blinn. Simulation of wrinkled faces. In *Proceedings of
            SIGGRAPH '78, Computer Graphics*, pages 286–292, 1978.

[Bli84]   James F. Blinn. Homogeneous properties of second order surfaces,
            1984. course notes, ACM SIGGRAPH '87, Vol. 12, July 1984.

[BM65]   G. Birkhoff and S. MacLane. *A Survey of Modern Algebra*. MacMil-
            lan, New York, 3rd edition, 1965. Exercise 15, Section IX-3, p. 240;
            also corollary, Section IX-14, pp. 277–278.

[BN76]   James F. Blinn and Martin E. Newell. Texture and reflection in com-
            puter generated images. *Communications of the ACM*, 19(10):542–
            547, 1976.

[BO79]   J. L. Bentley and T. Ottmann. Algorithms for reporting and count-
            ing geometric intersections. *IEEE Transactions on Computers*, C-
            28(September):643–647, 1979.

[Bra82]   M Brady. Trajectory planning. In M. Brady, M. Hollerback, T.L.
            Johnson, T. Lozano-Perez, and M.T. Mason, editors, *Robot Motion:
            Planning and Control*. MIT Press, 1982.

[Bre65]   J.E. Bresenham. Algorithm for computer control of a digital plotter.
            IBM Systems Journal, 4(1):25–30, 1965.

[BS63]   Petr Beckmann and Andre Spizzichino. *The Scattering of Electro-
            magnetic Waves from Rough Surfaces*. MacMillan, 1963.

[BS86]   Eric Bier and Ken R. Sloan. Two-part texture mapping. *IEEE
            Computer Graphics and Applications*, 6(9):40–53, 1986.

[BT75]   Phong Bui-Tuong. Illumination for computer generated pictures.
            Communications of the ACM, 18(6):311–317, 1975.

[BW76]   N. Burtnyk and M. Wein. Interactive skeleton techniques for en-
           hancing motion dynamics in key frame animation. *Communications
           of the ACM*, 19:564–569, 1976.

[BW86]   G. Bishop and D.M. Weimar. Fast phong shading. *Computer Graph-
           ics*, 20(4):103–106, 1986.

[Car84]   Loren Carpenter. The a-buffer, an atialiased hidden surface method.
           In *Proceedings of SIGGRAPH '84, Computer Graphics*, pages 103–
           108, 1984.

[Cat74]   E.E. Catmull. A subdivision algorithm for computer display and
           curved surfaces, 1974. Ph.D. Dissertation.

[Cat75]   E.E. Catmull. Computer display of curved surfaces. In *Proceedings
           of the IEEE Conference on Computer Graphics, Pattern Recognition
           and Data Structures*, 1975.

[Cat78]   Edwin Catmull. A hidden-surface algorithm with anti-aliasing. In
           Proceedings of SIGGRAPH '78, pages 6–11, 1978.

[CCC87]   Robert L. Cook, , Loren Carpenter, and Edwin Catmull. The reyes
           image rendering architecture. In *Proceedings of SIGGRAPH '87*,
           pages 95–102, 1987.

[CCWG88]   M.F. Cohen, S.E. Chen, J.R. Wallace, and D.P. Greenberg. A
           progressive refinement approach to fast radiosity image generation.
           In *SIGGRAPH '88 Proceedings, Computer Graphics*, pages 75–84,
           1988.

[CF89]   N. Chin and S. Feiner. Near real-time object-precision shadow gen-
           eration using bsp trees. In *SIGGRAPH '89 Proceedings, Computer
           Graphics*, pages 99–106, 1989.

[CG85a]   R.J. Carey and D.P. Greenberg. Textures for realistic image syn-
           thesis. *Computers and Graphics*, 9(2):125–138, 1985.

[CG85b]   Michael Cohen and Donald Greenberg. The hemi-cube, a radiosity
           solution for complex environments. In *Proceedings of SIGGRAPH
           '85*, pages 31–40, 1985.

[CGIB86]   Michael F. Cohen, Donald P. Greenberg, David S. Immel, and
           Phillip J. Brock. An efficient radiosity approach for realistic image

synthesis. *IEEE Computer Graphics and Applications*, 6(3):26–35, 1986.

[Cha82]   Bernard Chazelle. A theorem on polygon cutting with applications. In *Proc. 23rd Annual IEEE Symp. on Foundations of Computer Science*, pages 339–349, 1982.

[Chi88]   Hiroaki Chiyokura. *Solid Modelling with DESIGNBASE*. Addision Wesley, 1988.

[CJ78]   M. Cyrus and Beck J. Generalized two- and three-dimensional clipping. *Computers and Graphics*, 3(1):23–28, 1978.

[Coo86]   Robert L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, 1986.

[Cox74]   H.S.M. Coxeter. *Projective Geometry*. University of Toronto Press, Toronto, 1974.

[CPC84]   Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of SIGGRAPH '84, Computer Graphics*, pages 137–145, 1984.

[Cro77a]   Franklin C. Crow. The aliasing problem in computer-generated shaded images. *Communications of the ACM*, 20(11):799–805, 1977.

[Cro77b]   Franklin C. Crow. Shadow algorithm for computer graphics. In *Proceedings of SIGGRAPH '77, Computer Graphics*, pages 242–248, 1977.

[Cro81]   Franklin C. Crow. A comparison of antialiasing techniques. *Computer Graphics and Applications*, 1(1):40–48, 1981.

[Cro84]   Franklin C. Crow. Summed area tables for texture mapping. In *Proceedings of SIGGRAPH '84, Computer Graphics*, volume 18, pages 207–212, 1984.

[CT81]   Robert Cook and Kenneth Torrance. A reflectance model for computer graphics. *Computer Graphics*, 15(3), 1981.

[Dav54]   H Davis. The reflection of electromagnetic waves from a rough surface. In *Proceedings of the Institution of Electrical Engineers, v*, volume 101, pages 209–214, 1954.

[dB92]   Mark de Berg. *Efficient Algorithms for Ray Shooting and Hidden Surface Removal.* PhD thesis, Rijksuniversiteit te Utrecht, Nederlands, 1992.

[Dév93]   Ferenc Dévai. *Computational Geometry and Image Synthesis.* PhD thesis, Computer and Automation Institute, Hungarian Academy of Sciences, Budapest, Hungary, 1993.

[DM87]   B. P. Demidovich and I. A. Maron. *Computational Mathematics.* MIR Publishers, Moscow, 1987.

[DRSK92]   B. Dobos, P. Risztics, and L. Szirmay-Kalos. Fine-grained parallel processing of scan conversion with i860 microprocessor. In *7th Symp. on Microcomputer Appl.*, Budapest, 1992.

[Duf79]   Tom Duff. Smoothly shaded rendering of polyhedral objects on raster displays. In *Proceedings of SIGGRAPH '79, Computer Graphics*, 1979.

[Duv90]   V. Duvanenko. Improved line segment clipping. *Dr. Dobb's Journal*, july, 1990.

[EWe89]   R.A. Earnshaw and B. Wyvill (editors). *New Advances in Computer Graphics.* Springer-Verlag, Tokyo, 1989.

[Far88]   G. Farin. *Curves and Surfaces for Computer Aided Geometric Design.* Academic Press, New York, 1988.

[FFC82]   Alain Fournier, Don Fussel, and Loren C. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.

[FG85]   Cohen. Michael F. and Donald B. Greenberg. The hemi-cube: A radiosity solution for complex environments. In *Proceedings of SIGGRAPH '85, Computer Graphics*, pages 31–40, 1985.

[FKN80]   Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priory tree structures. In *Proceedings of SIGGRAPH '80*, pages 124–133, 1980.

[FLC80]   E.A. Feibush, M. Levoy, and R.L. Cook. Syntetic texturing using digital filters. In *SIGGRAPH '80 Proceedings, Computer Graphics*, pages 294–301, 1980.

[FP81] H. Fuchs and J. Poulton. Pixel-planes: A vlsi-oriented design for a raster graphics engine. *VLSI Design*, 3(3):20–28, 1981.

[Fra80] William Randolph Franklin. A linear time exact hidden surface algorithm. In *Proceedings of SIGGRAPH '80, Computer Graphics*, pages 117–123, 1980.

[FS75] R. Floyd and L. Steinberg. An adaptive algorithm for spatial gray scale. In *Society for Information Display 1975 Symposium Digest of Tecnical Papers*, page 36, 1975.

[FTK86] Akira Fujimoto, Tanaka Takayuki, and Iwata Kansei. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

[FvD82] J.D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley,, Reading, Mass., 1982.

[FvDFH90] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, Mass., 1990.

[GCT86] D.P. Greenberg, M.F. Cohen, and K.E. Torrance. Radiosity: A method for computing global illumination. *The Visual Computer 2*, pages 291–297, 1986.

[Ge89] A.S. Glassner (editor). *An Introduction to Ray Tracing*. Academic Press, London, 1989.

[GH86] N. Greene and P. S. Heckbert. Creating raster omnimax images using the elliptically weighted average filter. *IEEE Computer Graphics and Applications*, 6(6):21–27, 1986.

[Gla84] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[Gou71] H. Gouraud. Computer display of curved surfaces. *ACM Transactions on Computers*, C-20(6):623–629, 1971.

[GPC82] M. Gangnet, P. Perny, and P. Coueignoux. Perspective mapping of planar textures. In *EUROGRAPHICS '82*, pages 57–71, 1982.

[Gra72]   R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, (1):132–133, 1972.

[Gre84]   N. Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, 1984.

[Gre86]   N. Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, 1986.

[GS88]   Leonidas J. Guibas and Jorge Stolfi. Ruler, compass and computer. the design and analysis of geometric algorithms. In R. A. Earnshow, editor, *Theoretical Foundations of Computer Graphics and CAD*. Springer-Verlag, Berlin Heidelberg, 1988. NATO ASI Series, Vol. F40.

[GSS81]   S. Gupta, R. Sproull, and I. Sutherland. Filtering edges for gray-scale displays. In *SIGGRAPH '81 Proceedings, Computer Graphics*, pages 1–5, 1981.

[GTG84]   Cindy M. Goral, Kenneth E. Torrance, and Donald P. Greenberg. Modeling the interaction of light between diffuse surfaces. In *Proceedings of SIGGRAPH '84, Computer Graphics*, pages 213–222, 1984.

[Hal86]   R. Hall. A characterization of illumination models and shading techniques. *The Visual Computer 2*, pages 268–277, 1986.

[Hal89]   R. Hall. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York, 1989.

[Har69]   F. Harary. *Graph Theory*. Addison-Wesley, Massachusetts, 1969.

[Har87]   David Harel. *Algoritmics - The Spirit of Computing*. MacMillan, 1987.

[Hec86]   Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, 1986.

[Her91]   Ivan Herman. *The Use of Projective Geometry in Computer Graphics*. Spinger-Verlag, Berlin, 1991.

[Hit84]   Hitachi. *HD63484 ACRTC Advanced CRT Controller*. MSC Ver-
            triebs Gmbh, 1984.

[HKRSK91]   T. Horváth, E. Kovács, P. Risztics, and L. Szirmay-Kalos.
            Hardware-software-firmware decomposition of high-performace 3d
            graphics systems. In *6th Symp. on Microcomputer Appl.*, Budapest,
            1991.

[HMSK92]   T. Horváth, P. Márton, G. Risztics, and L. Szirmay-Kalos. Ray co-
            herence between sphere and a convex polyhedron. *Computer Graph-
            ics Forum*, 2(2):163–172, 1992.

[HRV92]   Tamás Hermann, Gábor Renner, and Tamás Várady. Mathematical
            techniques for interpolating surfaces with general topology. Tech-
            nical Report GML–1992/1, Computer and Automation Institute,
            Hungarian Academy of Sciences, Budapest, Hungary, 1992.

[HS67]   Hoyt C. Hottel and Adel F. Sarofin. *Radiative Transfer*. McGraw-
            Hill, New-York, 1967.

[HS79]   B.K.P. Horn and R.W. Sjoberg. Calculating the reflectance map.
            Applied Optics, 18(11):1170–1179, 1979.

[Hun87]   R.W. Hunt. *The Reproduction of Colour*. Fountain Press, Tolworth,
            England, 1987.

[ICG86]   David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A
            radiosity method for non-diffuse environments. In *Proceedings of
            SIGGRAPH '86, Computer Graphics*, pages 133–142, 1986.

[Ins86]   American National Standard Institute. Nomenclature and defini-
            tions for illumination engineering. Technical Report RP-16-1986,
            ANSI/IES, 1986.

[Int89a]   Intel. *i860 64-bit microprocessor: Hardware reference manual*. Intel
            Corporation, Mt. Prospect, IL, 1989.

[Int89b]   Intel. *i860 64-bit microprocessor: Programmer's reference manual*.
            Intel Corporation, Mt. Prospect, IL, 1989.

[ISO90]   ISO/IEC-9592. *Information processing systems - Computer
            graphics, Programmers's Hierarchical Interactive Graphics System
            (PHIGS)*. 1990.

[Jar73]   R. A. Jarvis. On the identification of the convex hull of a finite set
             of points in the plane. *Information Processing Letters*, (2):18–21,
             1973.

[JGMHe88]   Kenneth I. Joy, Charles W. Grant, Nelson L. Max, and Lansing
             Hatfield (editors). *Computer Graphics: Image Synthesis*. IEEE
             Computer Society Press, Los Alamitos, CA., 1988.

[Kaj82]   James T. Kajiya. Ray tracing parametric patches. In *Proceedings
             of SIGGRAPH '82, Computer Graphics*, pages 245–254, 1982.

[Kaj83]   James T. Kajiya. New techniques for ray tracing procedurally de-
             fined objects. In *Proceedings of SIGGRAPH '83, Computer Graph-
             ics*, pages 91–102, 1983.

[Kaj86]   James T. Kajiya. The rendering equation. In *Proceedings of SIG-
             GRAPH '86, Computer Graphics*, pages 143–150, 1986.

[KG79]   D.S. Kay and D. Greenberg. Transparency for computer synthesized
             pictures. In *SIGGRAPH '79 Proceedings, Computer Graphics*, pages
             158–164, 1979.

[KK75]   Granino A. Korn and Theresa M. Korn. *Mathematical Handbook for
             Scientist and Engineers*. McGraw-Hill, 1975.

[KKM29]   B. Knaster, C. Kuratowski, and S. Mazurkiewicz. Ein beweis des
             fixpunktsatzes für n-dimensionale simplexe. *Fund. Math.*, (14):132–
             137, 1929.

[KM76]   K. Kuratowski and A. Mostowski. *Set Theory*. North-Hollands,
             Amsterdam, The Netherlands, 1976.

[Knu73]   Donald Ervin Knuth. *The art of computer programming. Volume
             3 (Sorting and searching)*. Addison-Wesley, Reading, Mass. USA,
             1973.

[Knu76]   Donald Ervin Knuth. Big omicron and big omega and big theta.
             SIGACT News, 8(2):18–24, 1976.

[Kra89]   G. Krammer. Notes on the mathematics of the phigs output pipeline.
             Computer Graphics Forum, 8(8):219–226, 1989.

[Lam72]   John Lamperti. *Stochastic Processes*. Spinger-Verlag, 1972.

[Lan91]   Béla Lantos. *Robotok Irányitása*. Akadémiai Kiadó, Budapest, 1991. in Hungarian.

[LB83]   Y-D. Liang and B.A. Barsky. An analysis and algorithm for polygon clipping. *Communications of the ACM*, 26:868–877, 1983.

[LB84]   Y.D. Lian and B. Barsky. A new concept and method for line clipping. *ACM TOG*, 3(1):1–22, 1984.

[LRU85]   M.E. Lee, R.A. Redner, and S.P. Uselton. Statistically optimized sampling for distributed ray tracing. In *SIGGRAPH '85 Proceedings, Computer Graphics*, pages 61–67, 1985.

[LSe89]   Tom Lyche and Larry L. Schumaker (editors). *Mathematical Methods in Computer Aided Geometric Design*. Academic Press, San Diego, 1989.

[Män88]   M. Mäntylä. *Introduction to Solid Modeling*. Computer Science Press, Rockville, MD., 1988.

[Már94]   Gábor Márton. *Stochastic Analysis of Ray Tracing Algorithms*. PhD thesis, Department of Process Control, Budapest University of Technology, Budapest, Hungary, 1994. to appear, in Hungarian.

[Max46]   E.A. Maxwell. *Methods of Plane Projective Geometry Based on the Use of General Homogenous Coordinates*. Cambridge University Press, Cambridge, England, 1946.

[Max51]   E.A. Maxwell. *General Homogenous Coordinates in Space of Three Dimensions*. Cambridge University Press, Cambridge, England, 1951.

[McK87]   Michael McKenna. Worst-case optimal hidden-surface removal. *ACM Transactions on Graphics*, 6(1):19–28, 1987.

[Men75]   B. Mendelson. *Introduction to Topology, 3rd ed.* Allyn & Bacon, Boston, MA, USA, 1975.

[MH84]   G.S. Milller and C.R. Hoffman. Illumination and reflection maps: Simulated objects in simulated and real environment. In *Proceedings of SIGGRAPH '84*, 1984.

[Mih70]   Sz.G. Mihlin. *Variational methods in mathematical physics*. Nauka, Moscow, 1970.

[Mit87]   D.P. Mitchell. Generating aliased images at low sampling densities. In *SIGGRAPH '87 Proceedings, Computer Graphics*, pages 221–228, 1987.

[Moo66]   R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ., 1966.

[Moo77]   R. E. Moore. A test for existence of solutions to nonlinear systems. *SIAM J. Numer. Anal.*, 14(4 September):611–615, 1977.

[MRSK92]   G. Márton, P. Risztics, and L. Szirmay-Kalos. Quick ray-tracing exploiting ray coherence theorems. In *7th Symp. on Microcomputer Appl.*, Budapest, 1992.

[MTT85]   N Magnenat-Thallman and D. Thallman. *Principles of Computer Animation*. Springer, Tokyo, 1985.

[NNS72]   M.E. Newell, R.G. Newell, and T.L. Sancha. A new approach to the shaded picture problem. In *Proceedings of the ACM National Conference*, page 443, 1972.

[NS79]   W.M. Newman and R.F. Sproull. *Principles of Interactive Computer Graphics, Second Edition*. McGraw-Hill Publishers, New York, 1979.

[Nus82]   H.J. Nussbauer. *Fast Fourier Transform and Convolution Algorithms*. Spinger-Verlag, New York, 1982.

[Ode76]   J.T. Oden. *An Introduction to the Mathematical Theory of Finite Elements*. Wiley Interscience, New York, 1976.

[OM87]   Masataka Ohta and Mamoru Maekawa. Ray coherence theorem and constant time ray tracing algorithm. In T. L. Kunii, editor, *Computer Graphics 1987. Proc. CG International '87*, pages 303–314, 1987.

[PC83]   Michael Potmesil and Indranil Chakravarty. Modeling motion blur in computer generated images. In *Proceedings of SIGGRAPH '83, Computer Graphics*, pages 389–399, 1983.

[PD84]   Thomas Porter and Tom Duff. Compositing digital images. In *Proceedings of SIGGRAPH '84, Computer Graphics*, pages 253–259, 1984.

[Pea85]   Darwyn R. Peachey. Solid texturing of complex surfaces. In *Proceedings of SIGGRAPH '85, Computer Graphics*, pages 279–286, 1985.

[Per85]   Ken Perlin. An image synthetisizer. In *Proceedings of SIGGRAPH '85, Computer Graphics*, pages 287–296, 1985.

[PFTV88]   William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, USA, 1988.

[Pho75]   Bui Thong Phong. Illumination for computer generated images. *Communications of the ACM*, 18:311–317, 1975.

[PS85]   Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[PSe88]   Franz-Otto Peitgen and Dietmar Saupe (editors). *The Science of Fractal Images*. Pringer-Verlag, New York, 1988.

[RA89]   David F. Rogers and J. Alan Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill, New York, 1989.

[Ree81]   William T. Reeves. Inbetweening for computer animation utilizing moving point constraints. In *Proceedings of SIGGRAPH '81, Computer Graphics*, pages 263–269, 1981.

[Ree83]   William T. Reeves. Particle systems - a tecniques for modelling a class of fuzzy objects. In *Proceedings of SIGGRAPH '83, Computer Graphics*, pages 359–376, 1983.

[Rén81]   Alfréd Rényi. *Valószinűségszámitás*. Tankönyvkiadó, Budapest, 1981. in Hungarian.

[Req80]   Aristides A. G. Requicha. Representations for rigid solids: Theory, methods and systems. *Computing Surveys*, 12(4):437–464, 1980.

[Rog85]   D.F. Rogers. *Procedural Elements for Computer Graphics*. McGraw Hill, New York, 1985.

[RS63]   A. Rényi and R. Sulanke. Über die konvexe hülle von n zufällig gewählten punkten. *Z. Wahrscheinlichkeitstheorie*, 2:75–84, 1963.

[SA87]   T. W. Sederberg and D. C. Anderson. Steiner surface patches. *IEEE Computer Graphics and Applications*, 5(May):23–36, 1987.

[Sam89]   H. Samet. Implementing ray tracing with octrees and neighbor finding. *Computers and Graphics*, 13(4):445–460, 1989.

[Sch30]   Julius Pawel Schauder. Der fixpunktsatz in funktionalräumen. *Studia Mathematica*, (2):171–180, 1930.

[Sei88]   R. Seidel. Constrained delaunay triangulations and voronoi diagrams with obstacles. In *1978–1988, 10-Years IIG.*, pages 178–191. Inst. Inform. Process., Techn. Univ. Graz, 1988.

[SF73]   G. Strang and G. J. Fix. *An analysis of the finite element method.* Englewood Cliffs, Prentice Hall, 1973.

[SH74]   I.E. Sutherland and G.W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[SH81]   Robert Siegel and John R. Howell. *Thermal Radiation Heat Transfer.* Hemisphere Publishing Corp., Washington, D.C., 1981.

[Sho85a]   K. Shoemake. Animating rotation with quaternion curves. *Computer Graphics*, 19(3):245–254, 1985.

[Sho85b]   K. Shoemake. Animating rotation with quaternion curves. *Computer Graphics*, 16(3):157–166, 1985.

[Sim63]   G. F. Simmons. *Introduction to Topology and Modern Analysis.* McGraw-Hill, New York, 1963.

[SK88]   L. Szirmay-Kalos. *Árnyalási modellek a háromdimenziós raszter grafikában (Szakszemináriumi Füzetek 30).* BME, Folyamatszabályozási Tanszék, 1988. in Hungarian.

[SK93]   L. Szirmay-Kalos. Global element method in radiosity calculation. In *COMPUGRAPHICS '93*, Alvor, Portugal, 1993.

[SPL88]   M.Z. Shao, Q.S. Peng, and Y.D. Liang. A new radiosity approach by procedural refinements for realistic image synthesis. In *Proceedings of SIGGRAPH '86, Computer Graphics*, pages 93–101, 1988.

[SR92]   John Snyder and Barzel Ronen. Motion blur on graphics workstations. In David Kirk, editor, *Graphics Gems III*, pages 374–382. Academic Press, Boston, 1992.

[SSS74]   I.E. Sutherland, R.F. Sproull, and R.A. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, 1974.

[SSW86]   Marcel Samek, Chery Slean, and Hank Weghorst. Texture mapping and distortion in digital graphics. *Visual Computer*, 3:313–320, 1986.

[Tam92]   Filippo Tampieri. Accurate form-factor computation. In David Kirk, editor, *Graphics Gems III*, pages 329–333. Academic Press, Boston, 1992.

[Tex88]   Texas. *TMS34010: User's Guide*. Texas Instruments, 1988.

[Til80]   R. B. Tilove. Set membership classification: A unified approach to geometric intersection problems. *IEEE Transactions on Computers*, C-29(10):874–883, 1980.

[Tot85]   Daniel L. Toth. On ray tracing parametric surfaces. In *Proceedings of SIGGRAPH '85, Computer Graphics*, pages 171–179, 1985.

[Uli87]   R. Ulichney. *Digital Halftoning*. Mit Press, Cambridge, MA, 1987.

[Vár87]   Tamás Várady. Survey and new results in n-sided patch generation. In R. R. Martin, editor, *The Mathematics of Surfaces II*. Clarendon Press, Oxford, 1987.

[Vár91]   Tamás Várady. Overlap patches: a new scheme for interpolating curve networks with n-sided regions. *Computer Aided Geometric Design*, 8:7–27, 1991.

[WA77]   Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. In *Proceedings of SIGGRAPH '77, Computer Graphics*, pages 214–222, 1977.

[War69]   J.E. Warnock. A hidden line algorithm for halftone picture representation. Technical Report TR 4-15, Computer Science Department, University of Utah, Salt Lake City, Utah, 1969.

[Wat70]   G. Watkins. *A Real Time Hidden Surface Algorithm*. PhD thesis, Computer Science Department, University of Utah, Salt Lake City, Utah, 1970.

[Wat89]   A. Watt. *Fundamentals of Three-dimensional Computer Graphics.* Addision-Wesley, 1989.

[WCG87]   John R. Wallace, Michael F. Cohen, and Donald P. Greenberg. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. In *Proceedings of SIGGRAPH '87, Computer Graphics*, pages 311–324, 1987.

[WEH89]   John R. Wallace, K.A. Elmquist, and E.A. Haines. A ray tracing algorithm for progressive radiosity. In *Proceedings of SIGGRAPH '89, Computer Graphics*, pages 315–324, 1989.

[Whi80]   Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[Wil78]   Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of SIGGRAPH '78, Computer Graphics*, pages 270–274, 1978.

[Wil83]   Lance Williams. Pyramidal parametric. In *Proceedings of SIGGRAPH '83, Computer Graphics*, volume 17, pages 1–11, 1983.

[WMEe88]   M.J. Wozny, H.W. McLaughlin, and J.L. Encarnacao (editors). *Geometric Modeling for CAD Applications.* North Holland, Amsterdam, 1988.

[Wol90]   G. Wolberg. *Digital Image Warping.* IEEE Computer Society Press, Washington, DC., 1990.

[WS82]   G. Wyszecki and W. Stiles. *Color Science: Concepts and Methods, Quantitative Data and Formulae.* Wiley, New York, 1982.

[Yam88]   Fujio Yamaguchi. *Curves and Surfaces in Computer Aided Geometric Design.* Springer-Verlag, Berlin Heidelberg, 1988.

[Yel83]   John I.Jr. Yellot. Spectral consequences of photoreceptor sampling in the rhesus retina. *Science*, 221:382–385, 1983.

[YKFT84]   K. Yamaguchi, T. L. Kunii, K. Fujimura, and H. Toriya. Octree related data structures and algorithms. *IEEE Computer Graphics and Applications*, 4(1):53–59, 1984.

SUBJECT INDEX