GPU Pro^2

GPU Pro^2

Advanced Rendering Techniques

Edited by Wolfgang Engel

A K Peters, Ltd.
Natick, Massachusetts

Editorial, Sales, and Customer Service Office

A K Peters, Ltd.
5 Commonwealth Road, Suite 2C
Natick, MA 01760
www.akpeters.com

Library of Congress Cataloging-in-Publication Data

GPU Pro2 : advanced rendering techniques / edited by Wolfgang Engel.
p. cm.
Includes bibliographical references.
ISBN 978-1-56881-718-7 (hardback)
1. Rendering (Computer graphics) 2. Graphics processing units–Programming. 3. Computer graphics. 4. Real-time data processing. I. Engel, Wolfgang F. II. Title: GPU Pro 2.

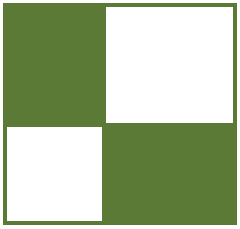
T385.G6885 2011
006.6–dc22

2010040134

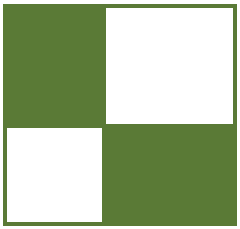
Front cover art courtesy of Lionhead Studios. All Fable®artwork appears courtesy of Lionhead Studios and Microsoft Corporation. "Fable" is a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries. ©2010 Microsoft Corporation. All Rights Reserved. Microsoft, Fable, Lionhead, the Lionhead Logo, Xbox, and the Xbox logo are trademarks of the Microsoft group of companies.

Back cover images are generated with CryENGINE 3 by Anton Kaplanyan. The Crytek Sponza model is publicly available courtesy of Frank Meinl.

Printed in India
14 13 12 11 10 9 8 7 6 5 4 3 2 1



Contents



Acknowledgments

The *GPU Pro: Advanced Rendering Techniques* book series covers ready-to-use ideas and procedures that can solve many of your daily graphics-programming challenges.

The second book in the series wouldn't have been possible without the help of many people. First, I would like to thank the section editors for the fantastic job they did. The work of Wessam Bahnassi, Sebastien St Laurent, Carsten Dachsbacher, Christopher Oat, and Kristof Beets ensured that the quality of the series meets the expectations of our readers.

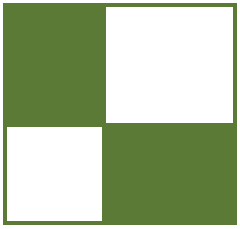
The great cover screenshots were taken from *Fable III*. I would like to thank Fernando Navarro from Microsoft Game Studios for helping us get the permissions to use those shots. You will find the *Fable III*-related article about morphological antialiasing in this book.

The team at A K Peters made the whole project happen. I want to thank Alice and Klaus Peters, Sarah Cutler, and the entire production team, who took the articles and made them into a book. Special thanks go out to our families and friends, who spent many evenings and weekends without us during the long book production cycle.

I hope you have as much fun reading the book as we had creating it.

—Wolfgang Engel

P.S. Plans for an upcoming *GPU Pro 3* are already in progress. Any comments, proposals, and suggestions are highly welcome (wolfgang.engel@gmail.com).



Web Materials

Example programs and source code to accompany some of the chapters are available at http://www.akpeters.com/gpupro. The directory structure closely follows the book structure by using the chapter number as the name of the subdirectory. You will need to download the DirectX August 2009 SDK.

General System Requirements

To use all of the files, you will need:

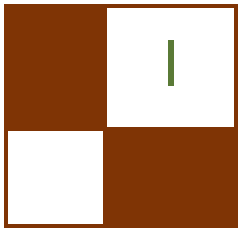
- The DirectX August 2009 SDK
- OpenGL 1.5-compatible graphics card
- A DirectX 9.0 or 10-compatible graphics card
- Windows XP with the latest service pack; some require VISTA or Windows 7
- Visual C++ .NET 2008
- 2048 MB RAM
- The latest graphics card drivers

Updates

Updates of the example programs will be periodically posted.

Comments and Suggestions

Please send any comments or suggestions to wolf@shaderx.com.



I Geometry Manipulation

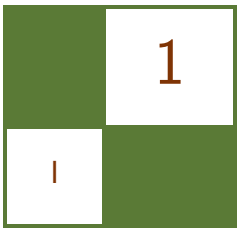
The "Geometry Manipulation" section of the book focuses on the ability of *graphics processing units* (GPUs) to process and generate geometry in exciting ways.

The article "Terrain and Ocean Rendering" looks at the tessellation related stages of DirectX 11, explains a simple implementation of terrain rendering, and implements the techniques from the *ShaderX*^{6} article "Procedural Ocean Effects" by László Szécsi and Khashayar Arman.

Jorge Jimenez, Jose I. Echevarria, Christopher Oat, and Diego Gutierrez present a method to add expressive and animated wrinkles to characters in the article "Practical and Realistic Facial Wrinkles Animation." Their system allows the animator to independently blend multiple wrinkle maps across regions of a character's face. When combined with traditional blend-target morphing for facial animation, this technique can produce very compelling results that enable virtual characters to be much more expressive in both their actions and dialog.

The article "Procedural Content Generation on GPU," by Aleksander Netzel and Pawel Rohleder, demonstrates the generating and rendering of infinite and deterministic heightmap-based terrain utilizing fractal Brownian noise calculated in real time on the GPU. Additionally it proposes a random tree distribution scheme that exploits previously generated terrain information. The authors use spectral synthesis to accumulate several layers of approximated fractal Brownian motion. They also show how to simulate erosion in real time.

—Wolfgang Engel



Terrain and Ocean Rendering with Hardware Tessellation

Xavier Bonaventura

Currently, one of the biggest challenges in computer graphics is the reproduction of detail in scenes. To get more realistic scenes you need high-detail models, which slow the computer. To increase the number of frames per second, you can use low-detail models; however, that doesn't seems realistic. The solution is to combine high-detail models near the camera and low-detail models away from the camera, but this is not easy.

One of the most popular techniques uses a set of models of different levels of detail and, in the runtime, changes them depending on their distance from the camera. This process is done in the CPU and is a problem because the CPU is not intended for this type of work—a lot of time is wasted sending meshes from the CPU to the GPU.

In DirectX 10, you could change the detail of meshes into the GPU by performing tessellation into the geometry shader, but it's not really the best solution. The output of the geometry shader is limited and is not intended for this type of serial work.

The best solution for tessellation is the recently developed tessellator stage in DirectX 11. This stage, together with the hull and the domain shader, allows the programmer to tessellate very quickly into the GPU. With this method you can send low-level detail meshes to the GPU and generate the missing geometry to the GPU depending on the camera distance, angle, or whatever you want.

In this article we will take a look at the new stages in DirectX 11 and how they work. To do this we will explain a simple implementation of terrain rendering and an implementation of water rendering as it appeared in *ShaderX*^{6} [Szécsi and Arman 08], but using these tools.

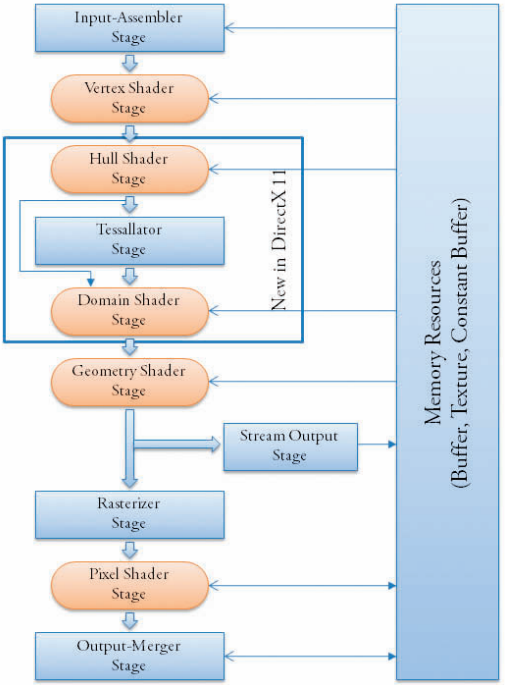


Figure 1.1. DirectX 11 pipeline.

1.1 DirectX 11 Graphics Pipeline

The DirectX 11 graphics pipeline [Microsoft] adds three new stages to the DirectX 10: the hull shader stage, tessellator stage, and domain shader stage (see Figure 1.1). The first and third are programmable and the second is configurable. These come after the vertex shader and before the geometry shader, and they are intended to do tessellation into the graphic card.

1.1.1 Hull Shader Stage

The hull shader stage is the first part of the tessellation block. The data used in it is new in DirectX 11, and it uses a new primitive topology called *control point patch list*. As its name suggests, it represents a collection of control points wherein the number in every patch can go from 1 to 32. These control points are required to define the mesh.

The output data in this stage is composed of two parts—one is the input control points that can be modified, and the other is some constant data that will be used in the tessellator and domain shader stages.

To calculate the output data there are two functions. The first is executed for every patch, and there you can calculate the tessellation factor for every edge of the patch and inside it. It is defined in the high-level shader language (HLSL) code and the attribute `[patchconstantfunc(''func_name'')]` must be specified.

The other function, the main one, is executed for every control point in the patch, and there you can manipulate this control point. In both functions, you have the information of all the control points in the patch; in addition, in the second function, you have the ID of the control point that you are processing.

An example of a hull shader header is as follows:

```
HS_CONSTANT_DATA_OUTPUT TerrainConstantHS(
  InputPatch<VS_CONTROL_POINT_OUTPUT, INPUT_PATCH_SIZE> ip,
  uint PatchID : SV_PrimitiveID )

[domain(''quad'')]
[partitioning(''integer'')]
[outputtopology(''triangle_cw'')]
[outputcontrolpoints(OUTPUT_PATCH_SIZE)]
[patchconstantfunc(''TerrainConstantHS'')]
HS_OUTPUT hsTerrain(
  InputPatch<VS_CONTROL_POINT_OUTPUT, INPUT_PATCH_SIZE> p,
  uint i : SV_OutputControlPointID,
  uint PatchID : SV_PrimitiveID )
```

For this example, we clarify some of the elements.

- `HS_CONSTANT_DATA_OUTPUT` is a struct and it must contain `SV_TessFactor` and `SV_InsideTessFactor`. Their types are dependent on `[domain(type_str)]`.
- `INPUT_PATCH_SIZE` is an integer and it must match with the `control point primitive`.
- `[domain(''quad'')]` can be either `''quad''`, `''tri''`, or `''isoline''`.
- `[partitioning(''integer'')]` can be either `''fractional_odd''`, `''fractional_even''`, `''integer''`, or `''pow2''`.
- `[outputtopology(''triangle_cw'')]` can be either `''line''`, `''triangle_cw''`, or `''triangle_ccw''`.
- `OUTPUT_PATCH_SIZE` will affect the number of times the main function will be executed.

1.1.2 Tessellator Stage

The tessellator stage is a part of the new pipeline wherein the programmer, setting some values, can change its behavior although he cannot program it. It is executed once per patch; the input data are the control points and the tessellation factors, which are the output from the hull shader. This stage is necessary to divide a quad, triangle, or line among many of them, depending on the tessellation factor and the type of partitioning defined (Figure 1.2). The tessellation factor on each edge defines how many divisions you want. The output is UV or UVW coordinates which go from 0 to 1 and define the position of new vertices relative to the patch. If the patch is a triangle, then UVW represents the position in barycentric coordinates. If the patch is a quad, then the UV coordinates are the position within the quad.

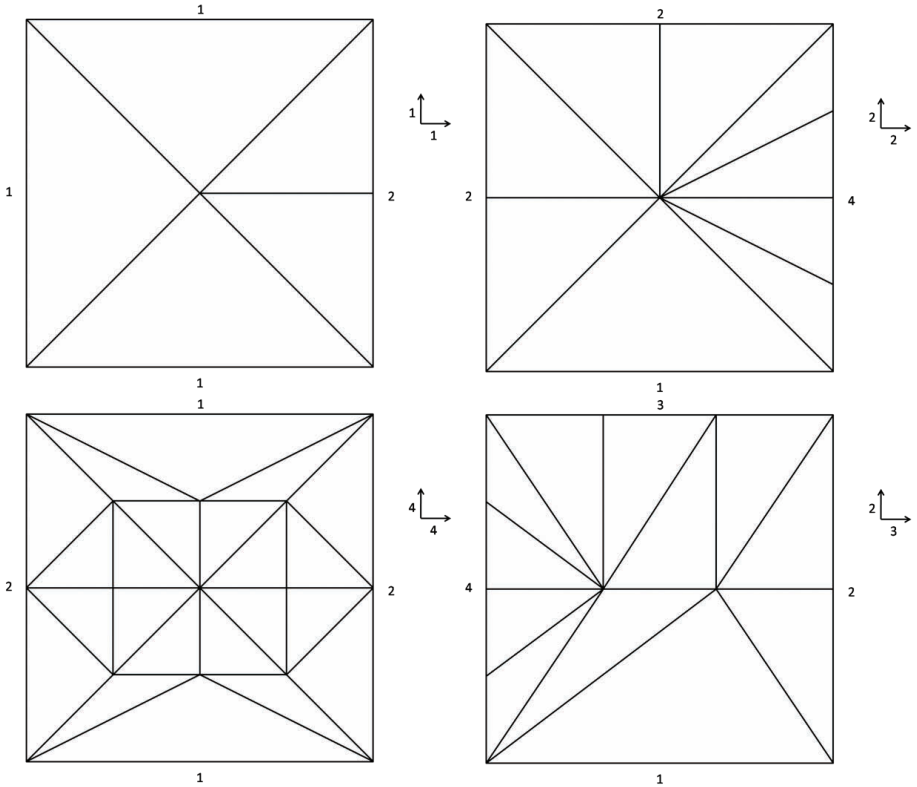


Figure 1.2. Tessellation in a quad domain: to define the tessellation on the edges, we start on the left edge and then we conitnue counterclockwise. To define the tessellation inside, first we input the horizontal and then the vertical tessellation factor.

1.1.3 Domain Shader Stage

The domain shader stage is the last stage of the tessellation. This part is executed once per UV coordinate generated in the tessellator stage, and it accesses the information from the hull shader stage output (control points and constant data) and UV coordinates. In this final stage, the aim is to calculate the final position of every vertex generated and all the associated information as normal, color, texture coordinate, etc.

Below is an example of a domain shader header:

```
[domain(''quad'')]
DS_OUTPUT dsTerrain(
  HS_CONSTANT_DATA_OUTPUT input,
  float2 UV : SV_DomainLocation,
  const OutputPatch<HS_OUTPUT, OUTPUT_PATCH_SIZE> patch )
```

The type of `SV_DomainLocation` can be different in other domain shaders. If the `[domain(_)]` is `''quad''` or `''isoline''`, its type is `float2`, but if the domain is `''tri''`, its type is `float3`.

1.2 Definition of Geometry

To tessellate the terrain and water, we need to define the initial geometry—to do this, we will divide a big quad into different patches. The tessellation can be

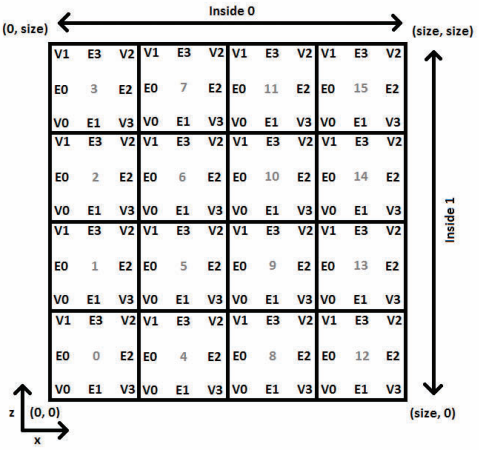


Figure 1.3. Division of the terrain into a grid: Vx and Ex represent vertices and edges, respectively, in every patch where x is the index to access. Inside 0 and Inside 1 represent the directions of the tessellation inside a patch.

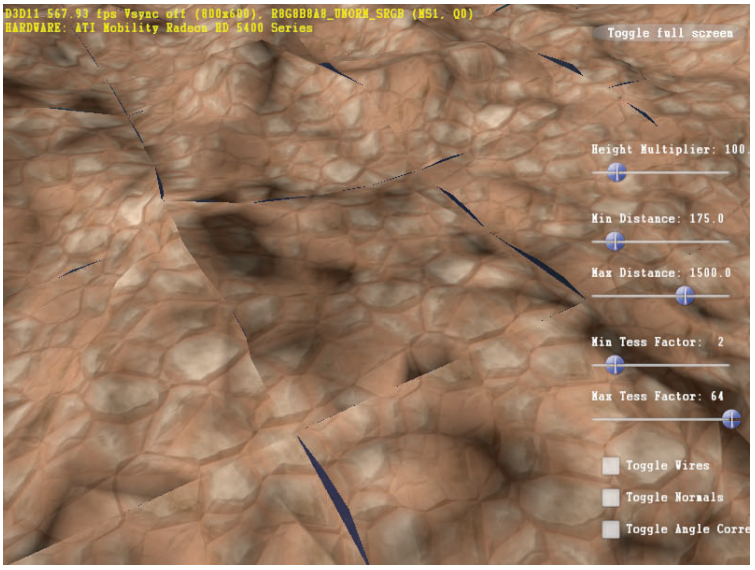


Figure 1.4. Lines between patches when tessellation factors are wrong.

applied to a lot of shapes, but we will use the most intuitive shape: a patch with four points. We will divide the terrain into patches of the same size, like a grid (Figure 1.3). For every patch, we will have to decide the tessellation factor on every edge and inside. It's very important that two patches that share the same edge have the same tessellation factor, otherwise you will see some lines between patches (Figure 1.4).

1.2.1 Tessellation Factor

In the tessellator stage you can define different kinds of tessellations (`fractional_even`, `fractional_odd`, `integer`, or `pow2`). In the terrain rendering we will use the integer, but in addition we will impose one more restriction: this value must be a power of two, to avoid a wave effect. This way, when a new vertex appears it will remain until the tessellation factor decreases again and its x- and z-values will not change. The only value that will change will be the y-coordinate, to avoid popping. In the ocean rendering we will not impose the power-of-two restriction because the ocean is dynamic and the wave effect goes unnoticed.

The tessellation factor ranges from 1 to 64 when the type of partitioning is integer, but the user has to define the minimum and the maximum values. We do not want the minimum of the tessellation factor to be 1 if there are few patches,

or the maximum to be 64 if the computer is slow. In the terrain rendering, the minimum and the maximum value must be powers of two.

We also have to define two distances between the camera and the point where we want to calculate the tessellation factor: the minimum and the maximum. When the distance is minimum the tessellation factor is maximum and when the distance is maximum the tessellation factor is minimum.

The tessellation factor will be 2^x where x is a number whose range will be from $\log_2$ MaxTessellation to $\log_2$ MinTessellation linearly interpolated between the minimum and the maximum distance. In terrain rendering, this x will be rounded to the nearest integer to get a power-of-two tessellation factor (see Equation (1.1)). In the ocean rendering, x will not be rounded (see Equation (1.2)).

$$te(d) = \begin{cases} 2^{\max(te_{\log 2})}, & \text{for } d \leq \min(d), \\ 2^{\text{round}(\text{diff}(te_{\log 2})(1-\frac{d-\min(d)}{\text{diff}(d)})+\min(te_{\log 2}))}, & \text{for } \min(d) < d < \max(d), \\ 2^{\min(te_{\log 2})}, & \text{for } d \geq \max(d). \end{cases} \tag{1.1}$$

$$te(d) = \begin{cases} 2^{\max(te_{\log 2})}, & \text{for } d \leq \min(d), \\ 2^{\text{diff}(te_{\log 2})(1-\frac{d-\min(d)}{\text{diff}(d)})+\min(te_{\log 2})}, & \text{for } \min(d) < d < \max(d), \\ 2^{\min(te_{\log 2})}, & \text{for } d \geq \max(d), \end{cases} \tag{1.2}$$

where $\text{diff}(x) = \max(x) - \min(x)$ and d is the distance from the point where we want to calculate the tessellation factor to the camera. The distances defined by the user are $\min(d)$ and $\max(d)$ and $\min(te_{\log_2})$ and $\max(te_{\log_2})$ are the tessellation factors defined by the user. For the tessellation factors, we use the $\log_2$ in order to get a range from 0 to 6 instead of from 1 to 64. The final value $te(d)$ is calculated five times for every patch, using different distances—four for the edges and one for inside.

As we said before, when an edge is shared by two patches the tessellation factor must be the same. To do this we will calculate five different distances in every patch, one for each edge and one inside. To calculate the tessellation factor for each edge, we calculate the distance between the camera and the central point of the edge. This way, in two adjacent patches with the same edge, the distance at the middle point of this edge will be the same because they share the two vertices that we use to calculate it. To calculate the tessellation factor inside the patch in U and V directions, we calculate the distance between the camera position and the middle point of the patch.

1.3 Vertex Position, Vertex Normal, and Texture Coordinates

In the domain shader we have to reconstruct every final vertex and we have to calculate the position, normal, and texture coordinates. This is the part where the difference between terrain and ocean rendering is more important.

1.3.1 Terrain

In terrain rendering (see Figure 1.5) we can easy calculate the x- and z-coordinates with a single interpolation between the position of the vertices of the patch, but we also need the y-coordinate that represents the height of the terrain at every point and the texture coordinates. Since we have defined the terrain, to calculate the texture coordinates, we have only to take the final x- and z-positions and divide by the size of the terrain. This is because the positions of the terrain range from 0 to the size of the terrain, and we want values from 0 to 1 to match all the texture over it.

Once we have the texture coordinates, to get the height and the normal of the terrain in a vertex, we read the information from a heightmap and a normal map in world coordinates combined in one texture. To apply this information, we have to use mipmap levels or we will see some popping when new vertices appear. To reduce this popping we get the value from a texture in which the concentration of points is the same compared to the concentration in the area where the vertex is located. To do this, we linearly interpolate between the minimum and the maximum mipmap levels depending on the distance (see Equation (1.3)).

Four patches that share a vertex have to use the same mipmap level in that vertex to be coherent; for this reason, we calculate one mipmap level for each vertex in a patch. Then, to calculate the mipmap level for the other vertices, we have only to interpolate between the mipmap levels of the vertices of the patch, where $\text{diff}(x) = \max(x) - \min(x)$, M = MipmapLevel, and d is the distance from the point to the camera:

$$\text{Mipmap}(d) = \begin{cases} \min(M), & \text{for } d \leq \min(d), \\ \text{diff}(M)\frac{d-\min(d)}{\text{diff}(d)} + \min(M), & \text{for } \min(d) < d < \max(d), \\ \max(M), & \text{for } d \geq \max(d). \end{cases} \quad (1.3)$$

To calculate the minimum and the maximum values for the mipmap level variables, we use the following equations, where textSize is the size of the texture that we use for the terrain:

$$\min(\text{MipmapLevel}) = \log_2(\text{textSize}) - \log_2(\text{sqrtNumPatch} \cdot 2^{\max(te)})$$
$$\max(\text{MipmapLevel}) = \log_2(\text{textSize}) - \log_2(\text{sqrtNumPatch} \cdot 2^{\min(te)})$$

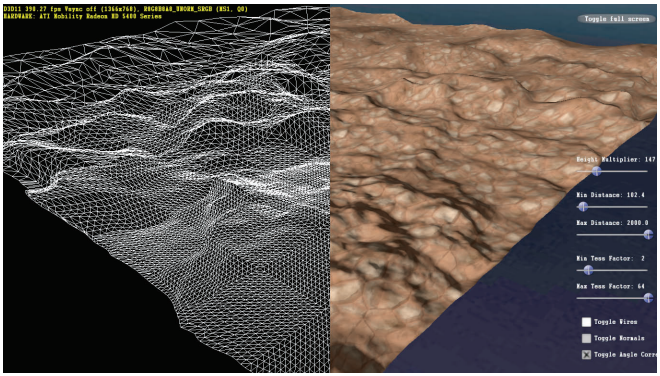


Figure 1.5. Terrain rendering.

We have to keep in mind that we use only squared textures with a power-of-two size. If the minimum value is less than 0, we use 0.

1.3.2 Ocean

Calculating the final vertex position in ocean rendering (see Figure 1.6) is more difficult than calculating terrain rendering. For this, we do not have a heightmap, and we have to calculate the final position depending on the waves and the position in the world coordinate space. To get real motion, we will use the technique explained in *ShaderX*[6] developed by Szécsi and Arman [Szécsi and Arman 08].

First, we have to imagine a single wave with a wavelength (λ), an amplitude (a), and a direction (k). Its velocity (v) can be represented by

$$v = \sqrt{\frac{g\lambda}{2\pi}}.$$

Then the phase (φ) at time (t) in a point (p) is

$$\varphi = \frac{2\pi}{\lambda}\left(p \cdot k + vt\right).$$

Finally, the displacement (s) to apply at that point is

$$s = a[-\cos\varphi, \sin\varphi].$$

An ocean is not a simple wave, and we have to combine all the waves to get a realistic motion:

$$p_\Sigma = p + \sum_{i=0}^{n} s(p, a_i, \lambda_i, k_i).$$

All a_i, λ_i, and k_i values are stored in a file.

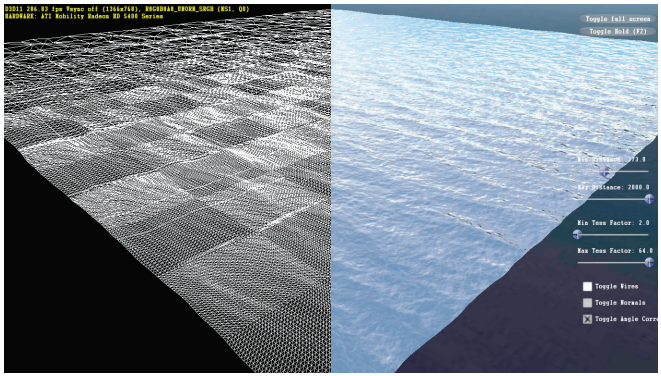


Figure 1.6. Ocean rendering.

At every vertex, we need to calculate the normal—it's not necessary to access a normal texture because it can be calculated by a formula. We have a parametric function where the parameters are the x- and z-coordinates that we have used to calculate the position; if we calculate the cross product of the two partial derivatives, we get the normal vector at that point:

$$N = \frac{\partial p_\Sigma}{\partial z} \times \frac{\partial p_\Sigma}{\partial x}.$$

1.4 Tessellation Correction Depending on the Camera Angle

So far we have assumed that the tessellation factor depends only on the distance from the camera; nevertheless, it's not the same if you see a patch from one direction or from another. For this reason, we apply a correction to the tessellation factor that we have calculated previously. This correction will not be applied to the final tessellation factor because we want the final one to be a power-of-two value; this correction will be applied to the unrounded value x that we use in 2^x to calculate the final tessellation factor.

The angle to calculate this correction will be the angle between the unit vector over an edge ($\widehat{e}$) and the unit vector that goes from the middle of this edge to the camera ($\widehat{c}$).

To calculate the correction, we will use this formula (see Figure 1.7):

$$\frac{\frac{\pi}{2} - \arccos\left(|\widehat{c} \cdot \widehat{e}|\right)}{\frac{\pi}{2}} \text{rank} + 1 - \frac{\text{rank}}{2}.$$

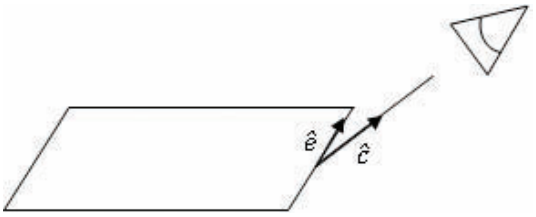


Figure 1.7. Angle correction.

The rank value is used to decide how important the angle is compared with the distance. The programmer can decide the value, but is advised to use values from 0 to 1 to reduce or increase the tessellation factor by 50%. If you decide to use a rank of 0.4 then the tessellation factor will be multiplied by a value between 0.8 and 1.2, depending on the angle.

To be consistent with this modification, we have to apply the correction to the value that we use to access the mipmap level in a texture. It is very important to understand that four patches that share the same vertex have the same mipmap value at that vertex. To calculate the angle of the camera at this point, we calculate the mean of the angles between the camera ($\widehat{c}$) and every vector over the edges ($\widehat{v0}$, $\widehat{v1}$, $\widehat{v2}$, $\widehat{v3}$) that share the point (see Figure 1.8):

$$\frac{\frac{\pi}{2} - \frac{\arccos\left(\left|\widehat{c}\cdot\widehat{v0}\right|\right)+\arccos\left(\left|\widehat{c}\cdot\widehat{v1}\right|\right)+\arccos\left(\left|\widehat{c}\cdot\widehat{v2}\right|\right)+\arccos\left(\left|\widehat{c}\cdot\widehat{v3}\right|\right)}{4}}{\frac{\pi}{2}}\text{rank} + 1 - \frac{\text{rank}}{2}.$$

We don't know the information about the vertex of the other patches for the hull shader, but these vectors can be calculated in the vertex shader because we know the size of the terrain and the number of patches.

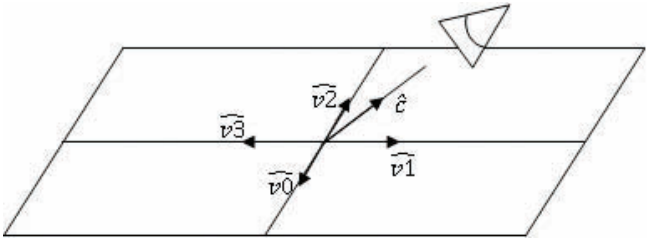


Figure 1.8. Mipmap angle correction.

1.5 Conclusions

As we have shown in this article, hardware tessellation is a powerful tool that reduces the information transfer from the CPU to the GPU. Three new stages added to the graphic pipeline allow great flexibility to use hardware tessellation advantageously. We have seen the application in two fields—terrain and water rendering—but it can be used in similar meshes.

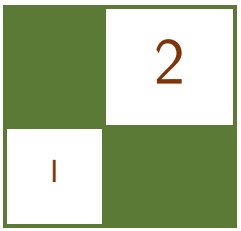
The main point to bear in mind is that we can use other techniques to calculate the tessellation factor, but we always have to be aware of the tessellation factor and mipmap levels with all the patches to avoid lines between them.

In addition, we have seen that it is better if we can use functions to represent a mesh like the ocean, because the resolution can be as high as the tessellation factor sets. If we use a heightmap, as we do for the terrain, it would be possible to not have enough information in the texture, and we would have to interpolate between texels.

Bibliography

[Microsoft] Microsoft. "Windows DirectX Grapics Documentation."

[Szécsi and Arman 08] László Szécsi and Khashayar Arman. *Procedural Ocean Effects*. Hingham, MA: Charles River Media, 2008.



Practical and Realistic Facial Wrinkles Animation

Jorge Jimenez, Jose I. Echevarria, Christopher Oat, and Diego Gutierrez

Virtual characters in games are becoming more and more realistic, with recent advances, for instance, in the techniques of skin rendering [d'Eon and Luebke 07, Hable et al. 09, Jimenez and Gutierrez 10] or behavior-based animation.[1] To avoid lifeless representations and to make the action more engaging, increasingly sophisticated algorithms are being devised that capture subtle aspects of the appearance and motion of these characters. Unfortunately, facial animation and the emotional aspect of the interaction have not been traditionally pursued with the same intensity. We believe this is an important aspect missing in games, especially given the current trend toward story-driven AAA games and their movie-like, real-time cut scenes.

Without even realizing it, we often depend on the subtleties of facial expression to give us important contextual cues about what someone is saying, thinking, or feeling. For example, a wrinkled brow can indicate surprise, while a furrowed brow may indicate confusion or inquisitiveness. In the mid-1800s, a French neurologist named Guillaume Duchenne performed experiments that involved applying electric stimulation to his subjects' facial muscles. Duchenne's experiments allowed him to map which facial muscles were used for different facial expressions. One interesting fact that he discovered was that smiles resulting from true happiness utilize not only the muscles of the mouth, but also those of the eyes. It is this subtle but important additional muscle movement that distinguishes a genuine, happy smile from an inauthentic or sarcastic smile. What we learn from this is that facial expressions are complex and sometimes subtle, but extraordinarily important in conveying meaning and intent. In order to allow artists to create realistic, compelling characters, we must allow them to harness the power of subtle facial expression.

[1] Euphoria NaturalMotion technology

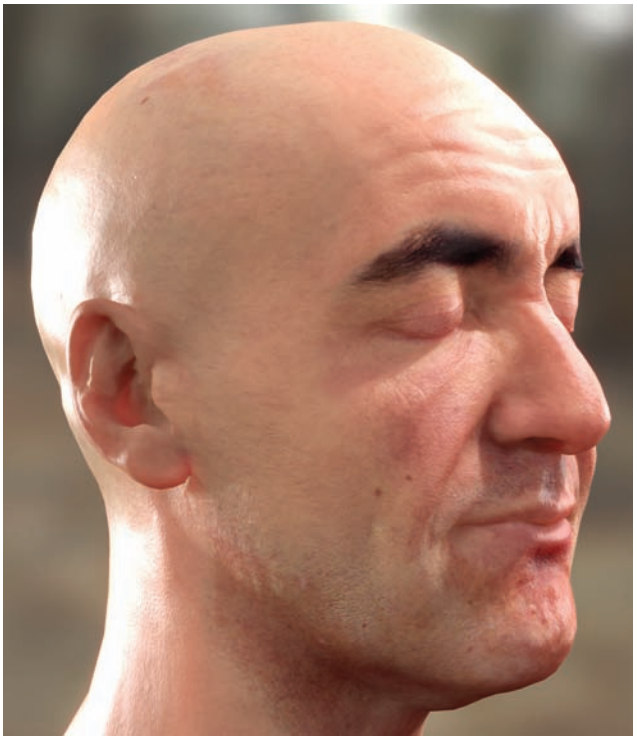


Figure 2.1. This figure shows our wrinkle system for a complex facial expression composed of multiple, simultaneous blend shapes.

We present a method to add expressive, animated wrinkles to characters' faces, helping to enrich stories through subtle visual cues. Our system allows the animator to independently blend multiple wrinkle maps across regions of a character's face. We demonstrate how combining our technique with state-of-the-art, real-time skin rendering can produce stunning results that enhance the personality and emotional state of a character (see Figures 2.1 and 2.2).

This enhanced realism has little performance impact. In fact, our implementation has a memory footprint of just 96 KB. Performance wise, the execution time of our shader is 0.31 ms, 0.1 ms, and 0.09 ms on a low-end GeForce 8600GT, mid-range GeForce 9800GTX+ and mid-high range GeForce 295GTX, respectively. Furthermore, it is simple enough to be added easily to existing rendering engines without requiring drastic changes, even allowing existing bump/normal textures to be reused, as our technique builds on top of them.

(a) Without wrinkles (b) With wrinkles

Figure 2.2. The same scene (a) without and (b) with animated facial wrinkles. Adding them helps to increase visual realism and conveys the mood of the character.

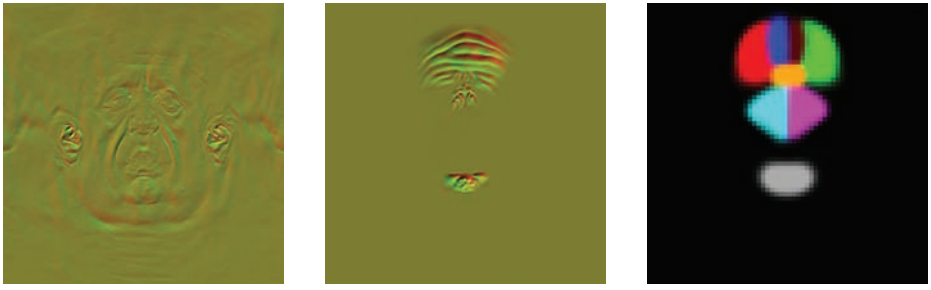
2.1 Background

Bump maps and normal maps are well-known techniques for adding the illusion of surface features to otherwise coarse, undetailed surfaces. The use of normal maps to capture the facial detail of human characters has been considered standard practice for the past several generations of real-time rendering applications. However, using static normal maps unfortunately does not accurately represent the dynamic surface of an animated human face. In order to simulate dynamic wrinkles, one option is to use length-preserving geometric constraints along with artist-placed wrinkle features to dynamically create wrinkles on animated meshes [Larboulette and Cani 04]. Since this method actually displaces geometry, the underlying mesh must be sufficiently tessellated to represent the finest level of wrinkle detail. A dynamic facial-wrinkle animation scheme presented recently [Oat 07] employs two wrinkle maps (one for stretch poses and one for compress poses), and allows them to be blended to independent regions of the face using artist-animated weights along with a mask texture. We build upon this technique, demonstrating how to dramatically optimize the memory requirements. Furthermore, our technique allows us to easily include more than two wrinkle maps when needed, because we no longer map negative and positive values to different textures.

2.2 Our Algorithm

The core idea of this technique is the addition of wrinkle normal maps on top of the base normal maps and blend shapes (see Figure 2.3 (left) and (center) for example maps). For each facial expression, wrinkles are selectively applied by using weighted masks (see Figure 2.3 (right) and Table 2.1 for the mask and weights used in our examples). This way, the animator is able to manipulate the wrinkles on a per-blend-shape basis, allowing art-directed blending between poses and expressions. We store a wrinkle mask per channel of a (RGBA) texture; hence, we can store up to four zones per texture. As our implementation uses eight zones, we require storing and accessing only two textures. Note that when the contribution of multiple blend shapes in a zone exceeds a certain limit, artifacts can appear in the wrinkles. In order to avoid this problem, we clamp the value of the summation to the $[0, 1]$ range.

While combining various displacement maps consists of a simple sum, combining normal maps involves complex operations that should be avoided in a time-constrained environment like a game. Thus, in order to combine the base



Base map Wrinkle map Mask map

Figure 2.3. The wrinkle map is selectively applied on top of the base normal map by using a wrinkle mask. The use of partial-derivative normal maps reduces this operation to a simple addition. The yellowish look is due to the encoding and storage in the R and G channels that this technique employs. Wrinkle-zone colors in the mask do not represent the actual channels of the mask maps, they are put together just for visualization purposes.

	Red	Green	Blue	Brown	Cyan	Magenta	Orange	Gray
Joy	1.0	1.0	0.2	0.2	0.0	0.0	0.0	0.0
Surprise	0.8	0.8	0.8	0.8	0.0	0.0	0.0	0.0
Fear	0.2	0.2	0.75	0.75	0.3	0.3	0.0	0.6
Anger	-0.6	-0.6	-0.8	-0.8	0.8	0.8	1.0	0.0
Disgust	0.0	0.0	-0.1	-0.1	1.0	1.0	1.0	0.5
Sad	0.2	0.2	0.75	0.75	0.0	0.0	0.1	1.0

Table 2.1. Weights used for each expression and zone (see color meaning in the mask map of Figure 2.3).

and wrinkle maps, a special encoding is used: partial-derivative normal maps [Acton 08]. It has two advantages over the conventional normal map encoding:

1. Instead of reconstructing the z-value of a normal, we just have to perform a vector normalization, saving valuable GPU cycles;

2. More important for our purposes, the combination of various partial-derivative normal maps is reduced to a simple sum, similar to combining displacement maps.

```
float3 WrinkledNormal(Texture2D<float2> baseTex,
                      Texture2D<float2> wrinkleTex,
                      Texture2D maskTex[2],
                      float4 weights[2],
                      float2 texcoord) {
  float3 base;
  base.xy = baseTex.Sample(AnisotropicSampler16, texcoord).gr;
  base.xy = -1.0 + 2.0 * base.xy;
  base.z = 1.0;

  #ifdef WRINKLES
  float2 wrinkles = wrinkleTex.Sample(LinearSampler,
                                      texcoord).gr;
  wrinkles = -1.0 + 2.0 * wrinkles;

  float4 mask1 = maskTex[0].Sample(LinearSampler, texcoord);
  float4 mask2 = maskTex[1].Sample(LinearSampler, texcoord);
  mask1 *= weights[0];
  mask2 *= weights[1];

  base.xy += mask1.r * wrinkles;
  base.xy += mask1.g * wrinkles;
  base.xy += mask1.b * wrinkles;
  base.xy += mask1.a * wrinkles;
  base.xy += mask2.r * wrinkles;
  base.xy += mask2.g * wrinkles;
  base.xy += mask2.b * wrinkles;
  base.xy += mask2.a * wrinkles;
  #endif

  return normalize(base);
}
```

Listing 2.1. HLSL code of our technique. We are using a linear instead of an anisotropic sampler for the wrinkle and mask maps because the low-frequency nature of their information does not require higher quality filtering. This code is a more readable version of the optimized code found in the web material.

This encoding must be run as a simple preprocess. Converting a conventional normal $n = (n_x, n_y, n_z)$ to a partial-derivative normal $n' = (n'_x, n'_y, n'_z)$ is done by using the following equations:

$$n'_x = \frac{n_x}{n_z} \qquad n'_y = \frac{n_y}{n_z}.$$

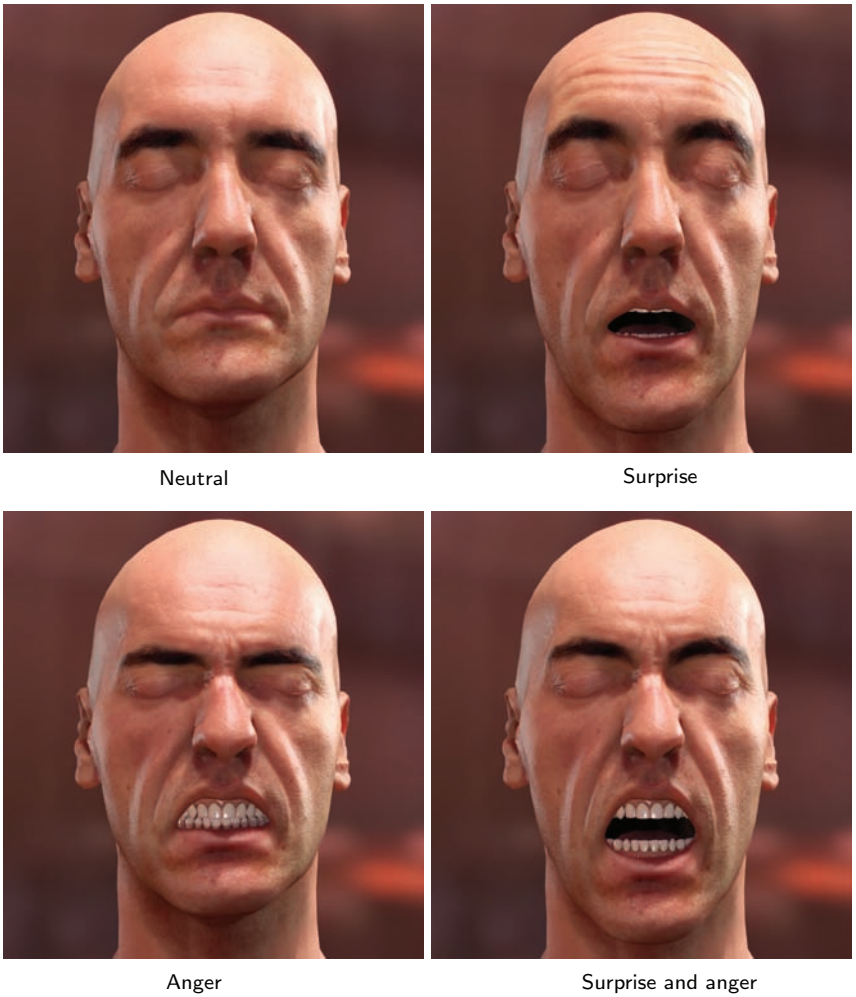


Figure 2.4. The net result of applying both surprise and anger expressions on top of the neutral pose is an unwrinkled forehead. In order to accomplish this, we use positive and negative weights in the forehead wrinkle zones, for the surprise and angry expressions, respectively.

In runtime, reconstructing a single partial-derivative normal n' to a conventional normal $\hat{n}$ is done as follows:

$$\begin{aligned} n &= (n'_x, n'_y, 1), \\ \hat{n} &= \frac{n}{\|n\|}. \end{aligned}$$

Note that in the original formulation of partial-derivative normal mapping there is a minus sign both in the conversion and reconstruction phases; removing it from both steps allows us to obtain the same result, with the additional advantage of saving another GPU cycle.

Then, combining different partial-derivative normal maps consists of a simple summation of their (x, y)-components before the normalization step. As Figure 2.3 reveals, expression wrinkles are usually low frequency. Thus, we can reduce map resolution to spare storage and lower bandwidth consumption, without visible loss of quality. Calculating the final normal map is therefore reduced to a summation of weighted partial-derivative normals (see Listing 2.1).

A problem with facial wrinkle animation is the modeling of compound expressions, through which wrinkles result from the interactions among the basic expressions they are built upon. For example, if we are surprised, the frontalis muscle contracts the skin, producing wrinkles in the forehead. If we then suddenly became angry, the corrugator muscles are triggered, expanding the skin in the forehead, thus causing the wrinkles to disappear. To be able to model these kinds of interactions, we let mask weights take negative values, allowing them to cancel each other. Figure 2.4 illustrates this particular situation.

2.2.1 Alternative: Using Normal Map Differences

An alternative to the use of partial-derivative normal maps for combining normal maps is to store differences between the base and each of the expression wrinkle maps (see Figure 2.5 (right)) in a manner similar to the way blend-shape interpolation is usually performed. As differences may contain negative values, we perform a scale-and-bias operation so that all values fall in the $[0, 1]$ range, enabling storage in regular textures:

$$d(x, y) = 0.5 + 0.5 \cdot (w(x, y) - b(x, y)),$$

where $w(x, y)$ is the normal at pixel (x, y) of the wrinkle map, and $b(x, y)$ is the corresponding value from the base normal map. When DXT compression is used for storing the differences map, it is recommended that the resulting normal be renormalized after adding the delta, in order to alleviate the artifacts caused by the compression scheme (see web material for the corresponding listing).

Base map Wrinkle map Difference map

Figure 2.5. We calculate a wrinkle-difference map by subtracting the base normal map from the wrinkle map. In runtime, the wrinkle-difference map is selectively added on top of the base normal map by using a wrinkle mask (see Figure 2.3 (right) for the mask). The grey color of the image on the right is due to the bias and scale introduced when computing the difference map.

Partial-derivative normal mapping has the following advantages over the differences approach:

- It can be a little bit faster because it saves one GPU cycle when reconstructing the normal, and also allows us to add only two-component normal derivatives instead of a full (x, y, z) difference; these two-component additions can be done two at once, in only one cycle. This translates to a measured performance improvement of $1.12x$ in the GeForce 8600GT, whereas we have not observed any performance gain in either the GeForce 9800GTX+ nor in the GeForce 295GTX .
- It requires only two channels to be stored vs. the three channels required for the differences approach. This provides higher quality because 3Dc can be used to compress the wrinkle map for the same memory cost.

On the other hand, the differences approach has the following advantages over the partial-derivative normal mapping approach:

- It uses standard normal maps, which may be important if this cannot be changed in the production pipeline.
- Partial-derivative normal maps cannot represent anything that falls outside of a $45°$ cone around $(0, 0, 1)$. Nevertheless, in practice, this problem proved to have little impact on the quality of our renderings.

The suitability of each approach will depend on both the constraints of the pipeline and the characteristics of the art assets.

2.3 Results

For our implementation we used DirectX 10, but the wrinkle-animation shader itself could be easily ported to DirectX 9. However, to circumvent the limitation that only four blend shapes can be packed into per-vertex attributes at once, we used the DirectX 10 stream-out feature, which allows us to apply an unlimited number of blend shapes using multiple passes [Lorach 07]. The base normal map has a resolution of 2048×2048, whereas the difference wrinkle and mask maps have a resolution of 256×256 and 64×64, respectively, as they contain only low-frequency information. We use 3Dc compression for the base and wrinkle maps, and DXT for the color and mask maps. The high-quality scanned head model and textures were kindly provided by XYZRGB, Inc., with the wrinkle maps created manually, adding the missing touch to the photorealistic look of the images. We used a mesh resolution of 13063 triangles, mouth included, which is a little step ahead of current generation games; however, as current high-end systems become mainstream, it will be more common to see such high polygon counts, especially in cinematics.

To simulate the subsurface scattering of the skin, we use the recently developed screen-space approach [Jimenez and Gutierrez 10, Jimenez et al. 10b], which transfers computations from texture space to screen space by modulating a convolution kernel according to depth information. This way, the simulation is reduced to a simple post-process, independent of the number of objects in the scene and easy to integrate in any existing pipeline. Facial-color animation is achieved using a recently proposed technique [Jimenez et al. 10a], which is based on *in vivo* melanin and hemoglobin measurements of real subjects. Another crucial part of our rendering system is the Kelemen/Szirmay-Kalos model, which provides realistic specular reflections in real time [d'Eon and Luebke 07]. Additionally, we use the recently introduced filmic tone mapper [Hable 10], which yields really crisp blacks.

Figure 2.6. Closeups showing the wrinkles produced by nasalis (nose), frontalis (forehead), and mentalis (chin) muscles.

Figure 2.7. Transition between various expressions. Having multiple mask zones for the forehead wrinkles allows their shape to change according to the animation.

	Shader execution time
GeForce 8600GT	0.31 ms
GeForce 9800GTX+	0.1 ms
GeForce 295GTX	0.09 ms

Table 2.2. Performance measurements for different GPUs. The times shown correspond specifically to the execution of the code of the wrinkles shader.

For the head shown in the images, we have not created wrinkles for the zones corresponding to the cheeks because the model is tessellated enough in this zone, allowing us to produce geometric deformations directly on the blend shapes.

Figure 2.6 shows different close-ups that allow appreciating the wrinkles added in detail. Figure 2.7 depicts a sequential blending between compound expressions, illustrating that adding facial-wrinkle animation boosts realism and adds mood to the character (frames taken from the movie are included in the web material).

Table 2.2 shows the performance of our shader using different GPUs, from the low-end GeForce 8600GT to the high-end GeForce 295GTX. An in-depth examination of the compiled shader code reveals that the wrinkle shader adds a per-pixel arithmetic instruction/memory access count of 9/3. Note that animating wrinkles is useful mostly for near-to-medium distances; for far distances it can be progressively disabled to save GPU cycles. Besides, when similar characters share the same (u, v) arrangement, we can reuse the same wrinkles, further improving the use of memory resources.

2.4 Discussion

From direct observation of real wrinkles, it may be natural to assume that shading could be enhanced by using techniques like ambient occlusion or parallax occlusion mapping [Tatarchuk 07]. However, we have found that wrinkles exhibit very little to no ambient occlusion, unless the parameters used for its generation are pushed beyond its natural values. Similarly, self-occlusion and self-shadowing can be thought to be an important feature when dealing with wrinkles, but in practice we have found that the use of parallax occlusion mapping is most often unnoticeable in the specific case of facial wrinkles.

Furthermore, our technique allows the incorporation of additional wrinkle maps, like the lemon pose used in [Oat 07], which allows stretching wrinkles already found in the neutral pose. However, we have not included them because they have little effect on the expressions we selected for this particular character model.

2.5 Conclusion

Compelling facial animation is an extremely important and challenging aspect of computer graphics. Both games and animated feature films rely on convincing characters to help tell a story, and a critical part of character animation is the character's ability to use facial expression. We have presented an efficient technique for achieving animated facial wrinkles for real-time character rendering. When combined with traditional blend-target morphing for facial animation, our technique can produce very compelling results that enable virtual characters to accompany both their actions and dialog with increased facial expression. Our system requires very little texture memory and is extremely efficient, enabling true emotional and realistic character renderings using technology available in widely adopted PC graphics hardware and current generation game consoles.

2.6 Acknowledgments

Jorge would like to dedicate this work to his eternal and most loyal friend Kazán. We would like to thank Belen Masia for her very detailed review and support, Wolfgang Engel for his editorial efforts and ideas to improve the technique, and Xenxo Alvarez for helping to create the different poses. This research has been funded by a Marie Curie grant from the Seventh Framework Programme (grant agreement no.: 251415), the Spanish Ministry of Science and Technology (TIN2010-21543), and the Gobierno de Aragón (projects OTRI 2009/0411 and CTPP05/09). Jorge Jimenez was additionally funded by a grant from the Gobierno de Aragón. The authors would also like to thank XYZRGB Inc. for the high-quality head scan.

Bibliography

[Acton 08] Mike Acton. "Ratchet and Clank Future: Tools of Destruction Technical Debriefing." Technical report, Insomniac Games, 2008.

[d'Eon and Luebke 07] Eugene d'Eon and David Luebke. "Advanced Techniques for Realistic Real-Time Skin Rendering." In *GPU Gems 3*, edited by Hubert Nguyen, Chapter 14, pp. 293–347. Reading, MA: Addison Wesley, 2007.

[Hable et al. 09] John Hable, George Borshukov, and Jim Hejl. "Fast Skin Shading." In *ShaderX*^{7}, edited by Wolfgang Engel, Chapter II.4, pp. 161–173. Hingham, MA: Charles River Media, 2009.

[Hable 10] John Hable. "Uncharted 2: HDR Lighting." Game Developers Conference, 2010.

[Jimenez and Gutierrez 10] Jorge Jimenez and Diego Gutierrez. "Screen-Space Subsurface Scattering." In *GPU Pro*, edited by Wolfgang Engel, Chapter V.7. Natick, MA: A K Peters, 2010.

[Jimenez et al. 10a] Jorge Jimenez, Timothy Scully, Nuno Barbosa, Craig Donner, Xenxo Alvarez, Teresa Vieira, Paul Matts, Veronica Orvalho, Diego Gutierrez,

and Tim Weyrich. "A Practical Appearance Model for Dynamic Facial Color." *ACM Transactions on Graphics* 29:6 (2010), Article 141.

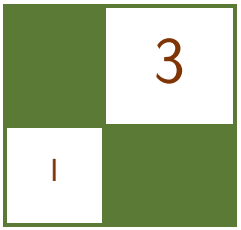
[Jimenez et al. 10b] Jorge Jimenez, David Whelan, Veronica Sundstedt, and Diego Gutierrez. "Real-Time Realistic Skin Translucency." *IEEE Computer Graphics and Applications* 30:4 (2010), 32–41.

[Larboulette and Cani 04] C. Larboulette and M. Cani. "Real-Time Dynamic Wrinkles." In *Proc. of the Computer Graphics International*, pp. 522–525. Washington, DC: IEEE Computer Society, 2004.

[Lorach 07] T. Lorach. "DirectX 10 Blend Shapes: Breaking the Limits." In *GPU Gems 3*, edited by Hubert Nguyen, Chapter 3, pp. 53–67. Reading, MA: Addison Wesley, 2007.

[Oat 07] Christopher Oat. "Animated Wrinkle Maps." In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pp. 33–37. New York: ACM, 2007.

[Tatarchuk 07] Natalya Tatarchuk. "Practical Parallax Occlusion Mapping." In *ShaderX*^{5}, edited by Wolfgang Engel, Chapter II.3, pp. 75–105. Hingham, MA: Charles River Media, 2007.



Procedural Content Generation on the GPU

Aleksander Netzel and Pawel Rohleder

3.1 Abstract

This article emphasizes on-the-fly procedural creation of content related to the video games industry. We demonstrate the generating and rendering of infinite and deterministic heightmap-based terrain utilizing fractal Brownian noise calculated in real time on a GPU. We take advantage of a thermal erosion algorithm proposed by David Cappola, which greatly improves the level of realism in heightmap generation. In addition, we propose a random tree distribution algorithm that exploits previously generated terrain information. Combined with the natural-looking sky model based on Rayleigh and Mie scattering, we achieved very promising quality results at real-time frame rates. The entire process can be seen in our DirectX10-based demo application.

3.2 Introduction

Procedural content generation (PCG) refers to the wide process of generating media algorithmically. Many existing games use PCG techniques to generate a variety of content, from simple, random object placement over procedurally generated landscapes to fully automatic creation of weapons, buildings, or AI enemies. Game worlds tend to be increasingly rich, which requires a lot of effort that we can minimize by utilizing PCG techniques. One of the basic PCG techniques in real-time computer graphics applications is the heightmap-based terrain generation [Olsen 04].

3.3 Terrain Generation and Rendering

Many different real-time, terrain-generation techniques have been developed over the last few years. Most of them utilize procedurally generated noise for creating a heightmap. The underlying-noise generation method should be fast enough to get at least close to real time and generate plausible results. The most interesting technique simulates 1/f noise (called "pink noise"). Because this kind of noise occurs widely in nature, it can be easily implemented on modern GPUs and has good performance/speed ratio. In our work, we decided to use the approximation of fractal Brownian motion (fBm).

Our implementation uses spectral synthesis, which accumulates several layers of noise together (see Figure 3.1). The underlying-noise generation algorithm is simple Perlin noise, which is described in [Green 05]. Although its implementation relies completely on the GPU, it is not fast enough to be calculated with every frame because of other procedural generation algorithms. We therefore used a system in which everything we need is generated on demand. The terrain is divided into smaller nodes (the number of nodes has to be a divider of heightmap size, so that there won't be any glitches after generation), and the camera is placed in the center of all nodes. Every node that touches the center node has a bounding box. Whenever collision between the camera and any of the bounding boxes is detected, a new portion of the heightmap is generated (along with other procedural content). Based on the direction of the camera collision and the position of nodes in world space, we determine new UV coordinates for noise

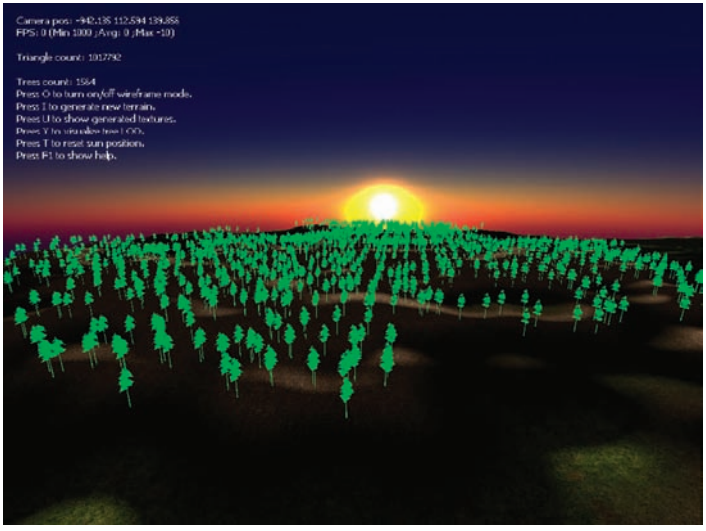


Figure 3.1. Procedurally generated terrain with random tree distribution.

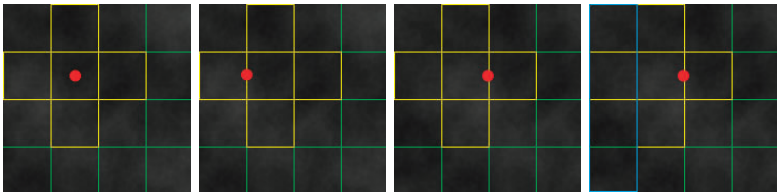


Figure 3.2. Red: camera, yellow: AABB, green: different patches in the grid. (a) Camera in the middle. (b) Collision detected with AABB. (c) New generation of procedural content, new AABB. (d) New row that is generated.

generation. This operation guarantees, with respect to the heightmap generation algorithm, that terrain will be continuous, endless, and realistic.

We could also optimize processing power by half, knowing that at most there are two rows of heightmaps that should be regenerated. We simply copy specified rows or columns and generate one additional row/column via the pixel shader (in Figure 3.2, all patches with a blue outline). When a camera collision occurs in the corner where two AABBs overlap, we have to generate a new row and column of data. To overcome the problem with camera collision and the newly generated AABB, we have to make the bounding boxes a little bigger or smaller, so there will be free space between them.

Our algorithm also utilizes a *level-of-detail* (LOD) system. Since we are assured of the relative positions of all nodes, we can precalculate all variations (there are only a small number) of index buffers. The center node has to have the lowest LOD (highest triangle count). Since we know that the camera will always be placed in the center node, we don't have to switch index buffers for different terrain LOD because the distance between the patch containing the camera and other patches is always the same (that's basically how our system works).

Another natural phenomenon we decided to simulate is erosion. Erosion is defined as the transporting and depositing of solid materials elsewhere due to wind, water, gravity, or living organisms. The simplest type of erosion is thermal erosion [Marak 97], wherein temperature changes cause small portions of the materials to crumble and pile up at the bottom of an incline. The algorithm is iterative and is as follows for each iteration: for every terrain point that has an altitude higher than the given threshold value (called *talus angle* (T)), some of its material will be moved to neighboring points. In the case that many of the neighbors' heights are above the talus angle, material has to be properly distributed.

The implementation is fairly simple. We compare every point of the heightmap with the heights of neighboring points and calculate how much material has to be moved. Since pixel shader limitation restricts us from scattering data

(before UAVs in DirectX11), we use an approach proposed by David Cappola [Capolla 08]. To overcome pixel shader limitation, we need to make two passes. In the first pass, we evaluate the amount of material that has to be moved from a given point to its neighbors; in the second pass, we actually accumulate material, using information provided by the first pass.

To improve the visual quality of our terrain, we generate a normal map, which is pretty straightforward. We use a given heightmap and apply the Sobel filter to it. Using the sun position, we can calculate shading (Phong diffuse). Since we target DX10 and higher we use texture arrays to render terrain with many textures [Dudask 07].

Collision detection is managed by copying a GPU-generated heightmap onto an offline surface and reading the height data array. Then we can check for collision using a standard method on the CPU and determine an accurate position inside an individual terrain quad using linear interpolation. Or, without involving the CPU, we could render to a 1×1 render target instead; however, since our tree placement technique is not "CPU free" (more on that later), we need the possibility of being able to check for height/collision without stressing the GPU.

3.4 Environmental Effects

Since we want to simulate a procedurally-rich natural environment, we have to be able to render more than just terrain with grass texture—we want plants. In the natural environment, plant distribution is based on many conditions such as soil quality, exposure (wind and sun), or the presence of other plants. In our work, we used a heightmap-based algorithm to determine tree placement along generated terrain.

Our tree-placement technique consists of two separate steps. First, we generate a tree-density map, which characterizes tree placement. The next step involves rendering instanced tree meshes. We use a density map to build a stream with world matrices. The height and slope of the terrain are the most common factors on which plant growth is dependent; both are easy to calculate in our case since we already have a heightmap and a normal map.

Using a simple pixel shader, we calculate density for every pixel as a mix of slope, height, and some arbitrarily chosen parameters. As a result, we get texture with tree distribution where value 1.0 corresponds to the maximum number of trees and 0.0 corresponds to no tree. To give an example, we present one of our versions (we tried many) in Listing 3.1 and the result in Figure 3.3. There are no strict rules on how to implement the shader. Anything that suits your needs is acceptable.

When the tree density map is ready to go, we have to build a stream with world matrices for instancing. Because the tree-density map texel can enclose a huge range in world space, we cannot simply place one tree per texel because the

```
float p = 0.0f;
//Calculate slope.
float f_slope_range = 0.17;
float f_slope_min = 1.0 - f_slope_range;
float3 v_normal = g_Normalmap.Sample( samClamp, IN.UV ).xyz * 2.0
                  - 1.0;
float f_height = g_Heightmap.Sample( samClamp, IN.UV ).x * 2.0
                 - 1.0;
float f_slope = dot( v_normal, float3(0,1,0) );
f_slope = saturate(f_slope - f_slope_min);
float f_slope_val = smoothstep(0.0, f_slope_range, f_slope);
//Get relative height.
float f_rel_height_threshold = 0.002;
float4 v_heights = 0;

v_heights.x = g_Heightmap.Sample( samClamp, IN.UV
              + float2( -1.0 / f_HM_size, 0.0) ); //Left
v_heights.y = g_Heightmap.Sample( samClamp, IN.UV
              + float2(  1.0 / f_HM_size, 0.0) ); //Right
v_heights.z = g_Heightmap.Sample( samClamp, IN.UV
              + float2(  0.0, -1.0 / f_HM_size) ); //Top
v_heights.w = g_Heightmap.Sample( samClamp, IN.UV
              + float2(  0.0,  1.0 / f_HM_size) ); //Down

v_heights = v_heights * 2.0 - 1.0;
v_heights = abs(v_heights - f_height);
v_heights = step(v_heights, f_rel_height_t);
p = dot(f_slope_val, v_heights) * 0.25;

return p;
```

Listing 3.1. Tree-density map pixel shader.

generated forest will be too sparse. To solve this issue, we have to increase the trees-per-texel ratio; therefore, we need one more placement technique.

We assign each tree type a different radius that determines how much space this type of tree owns (in world space). It can be compared to the situation in a real environment when bigger trees take more resources and prevent smaller trees from growing in their neighborhood. Also, we want our trees to be evenly but randomly distributed across a patch corresponding to one density map texel.

Our solution is to divide the current patch into a grid wherein each cell size is determined by the biggest tree radius in the whole patch. The total number of cells is a mix of the density of the current texel and the space that the texel encloses in world space. In the center of every grid cell, we place one tree and move its position using pseudorandom offset within the grid to remove repetitive patterns.

(a) (b) (c)

Figure 3.3. (a) Heightmap, (b) normal map, and (c) tree-density map.

Using camera frustum, we process only visible texels to form a density map. Based on the distance to the camera, the tree's LOD is determined so that only trees close to the camera will be rendered with a complete list of faces.

After all these steps, we have a second stream prepared and we are ready to render all the trees. We also input the maximum number of trees that we are about to render because there can easily become too many, especially when we have a large texel-to-world-space ratio. Care has to be taken with this approach since we cannot simply process a tree-density map row by row; we have to process texels close to the camera first. If we don't, we might use all "available" trees for the farthest patches, and there won't be any close to the camera. In this situation, we may use billboards for the farthest trees, or use a smooth transition into the fog color.

The last part of our procedural generation is the Rayleigh-Mie atmospheric scattering simulation [West 08, O'Neil 05]. Our implementation follows the technique described in *GPU Gems 2*. We first calculate optical depth to use it as a lookup table for further generating Mie and Rayleigh textures. Mie and Rayleigh textures are updated in every frame (using rendering to *multiple render targets* (MRT)) and are then sampled during sky-dome rendering. This method is efficient, fast, and gives visually pleasing results.

3.5 Putting It All Together

In Figure 3.4 we present what our rendering loop looks like. As described earlier, we perform only one full generation of procedural content whenever a collision of the camera with one of the bounding boxes is detected.

When collision is detected, we transform camera position into UV position for heightmap generation. After generating the heightmap, we calculate the erosion and the normal map. The last part of this generation step is to calculate new AABB.

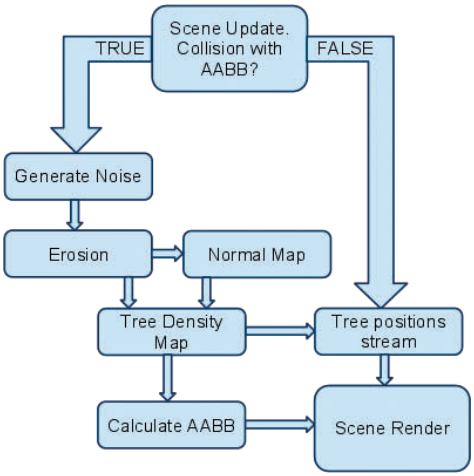
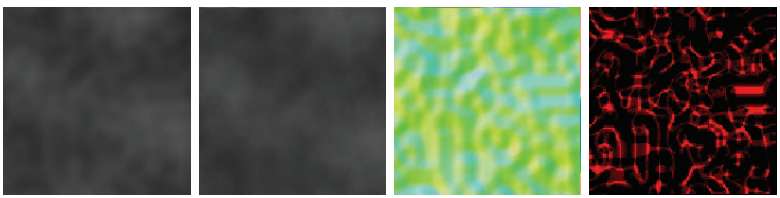


Figure 3.4. Rendering loop.

The tree-position stream is calculated with each frame since it depends on the camera orientation (see Figure 3.5).



(a) (b) (c) (d)

Figure 3.5. (a) Heightmap, (b) erosion map, (c) normal map, and (d) tree-density map.

3.6 Conclusions and Future Work

We implemented all of the techniques described in this article using Microsoft DirectX 10. All parameters controlling algorithm behavior can be changed during real time. Table 3.1 shows the minimum, maximum, and average number of *frames per second* (fps) in our framework, with 200 iterations of the erosion algorithm.

As we can see, the average number of frames is close to the number of maximum frames because usually only tree stream is generated. So a decrease of fps by a factor of 10 in the minimum is due to the generation of procedural content.

	NVIDIA GTX 260	ATI Radeon HD 5900
fps average	351	352
fps minimum	66	179
fps maximum	385	373

Table 3.1. Minimum, maximum, and average number of frames per second.

Therefore, we could procedurally generate content in every frame and keep up with real time, but our system doesn't require that.

The obvious optimization is to put the generation of the texture into another thread. For instance, erosion iterations can be divided into several frames.

Possibilities for developing further procedural generations are almost endless. We mention the techniques we will be developing in the near future. Tree-density mapping can be used not only for tree generation but also for other plant seeding systems like grass and bushes. Since we can assume that bushes can grow close to the trees, we can place some of them into the grid containing one tree. Of course, we must take into account that bushes are unlikely to grow directly in a tree's shadow. Also, the same density map can be used for grass placement. Since grass is likely to grow further away (because natural conditions for grass growth are less strict), we can blur the density map to get a grass density map. Therefore, without much more processing power, we can generate grass and bushes.

The next step could involve generating rivers and even whole cities, but cities put some conditions on generated terrain. Terrain under and around a city should be almost flat. The best solution is to combine artist-made terrain with procedural terrain. The only challenge is to achieve a seamless transition between the two.

Another interesting idea is to use other types of erosion (i.e., erosion described in *ShaderX*[7]) or different noise generators. For heightmap generation, since every function that returns height for a given x, y can be used, options are practically limitless.

For example, one might take advantage of the possibilities provided by the latest version of DirectX 11 Compute Shaders. It provides many new features that make procedural generation easier.

In conclusion, PCG offers the possibility to generate virtual worlds in a fast and efficient way. Many different techniques may be used to create rich environments, such as terrain with dense vegetation with a very small amount of artist/level designer work. Our application could be easily extended or integrated into an existing game editor. Since our techniques offer interactive frame rates it could be used for current games, like flight simulators or any other open-space game with a rich environment.

Bibliography

[Capolla 08] D. Capolla. "GPU Terrain." Available at http://www.m3xbox.com/GPU_blog, 2008.

[Dudask 07] B. Dudask. "Texture Arrays for Terrain Rendering." Available at http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/TextureArrayTerrain/doc/TextureArrayTerrain.pdf, 2007.

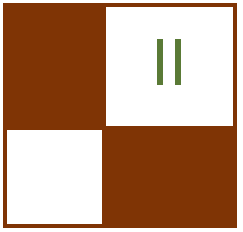
[Green 05] S. Green. "Implementing Improved Perlin Noise." In *GPU Gems 2*, edited by Hubert Nguyen, pp. 73–85. Reading, MA: Addison-Wesley, 2005.

[Marak 97] I. Marak. "Thermal Erosion." Availalbe at http://www.cg.tuwien.ac.at/hostings/cescg/CESCG97/marak/node11.html, 1997.

[Olsen 04] J. Olsen. "Real-Time Procedural Terrain Generation." Available at http://oddlabs.com/download/terrain_generation.pdf, 2004.

[O'Neil 05] S. O'Neil. *Accurate Atmospheric Scattering*. Reading, MA: Addison-Wesley, 2005.

[West 08] M. West. "Random Scattering: Creating Realistic Landscapes." Available at http://www.gamasutra.com/view/feature/1648/random_scattering_creating_.php, 2008.



II Rendering

In this section we cover new techniques in the field of real-time rendering. Every new generation of game or interactive application must push the boundaries of what is possible to render and simulate in real time in order to remain competitive and engaging. The articles presented here demonstrate some of the latest advancements in real-time rendering that are being employed in the newest games and interactive rendering applications.

The first article in the rendering section is "Pre-Integrated Skin Shading," by Eric Penner and George Borshukov. This article presents an interesting and very efficient shading model for rendering realistic skin. It can be evaluated entirely in a pixel shader and does not require extra rendering passes for blurring, thus making it a very scalable skin-rendering technique.

Our next article is "Implementing Fur in Deferred Shading," by Donald Revie. The popularity of deferred shading has increased dramatically in recent years. One of the limitations of working in a deferred-rendering engine is that techniques involving alpha blending, such as fur rendering, become difficult to implement. In this article we learn a number of tricks that enable fur to be rendered in a deferred-shading environment.

The third article in the rendering section is "Large-Scale Terrain Rendering for Outdoor Games," by Ferenc Pintér. This article presents a host of production-proven techniques that allow for large, high-quality terrains to be rendered on resource-constrained platforms such as current-generation consoles. This article provides practical tips for all areas of real-time terrain rendering, from the content-creation pipeline to final rendering.

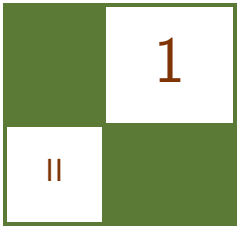
The fourth article in this section is "Practical Morphological Antialiasing," by Jorge Jimenez, Belen Masia, Jose I. Echevarria, Fernando Navarro, and Diego Gutierrez. The authors take a new, high-quality, antialiasing algorithm and demonstrate a highly optimized GPU implementation. This implementation is so efficient that it competes quite successfully with hardware-based antialiasing schemes in both performance and quality. This technique is particularly powerful because it provides a natural way to add antialiasing to a deferred-shading engine.

We conclude the section with Emil Persson's "Volume Decals" article. This is a practical technique to render surface decals without the need to generate

special geometry for every decal. Instead, the GPU performs the entire projection operation. The author shows how to use volume textures to render decals on arbitrary surfaces while avoiding texture stretching and shearing artifacts.

The diversity of the rendering methods described in this section represents the wide breadth of new work being generated by the real-time rendering community. As a fan of new and clever interactive rendering algorithms, reading and editing these articles has been a great joy. I hope you will enjoy reading them and will find them as useful and relevant as I do.

—Christopher Oat



Pre-Integrated Skin Shading

Eric Penner and George Borshukov

1.1 Introduction

Rendering realistic skin has always been a challenge in computer graphics. Human observers are particularly sensitive to the appearance of faces and skin, and skin exhibits several complex visual characteristics that are difficult to capture with simple shading models. One of the defining characteristics of skin is the way light bounces around in the dermis and epidermis layers. When rendering using a simple diffuse model, the light is assumed to immediately bounce equally in all directions after striking the surface. While this is very fast to compute, it gives surfaces a very "thin" and "hard" appearance. In order to make skin look more "soft" it is necessary to take into account the way light bounces around inside a surface. This phenomenon is known as *subsurface scattering*, and substantial recent effort has been spent on the problem of realistic, real-time rendering with accurate subsurface scattering.

Current skin-shading techniques usually simulate subsurface scattering during rendering by either simulating light as it travels through skin, or by gathering incident light from neighboring locations. In this chapter we discuss a different approach to skin shading: rather than gathering neighboring light, we pre-integrate the effects of scattered light. Pre-integrating allows us to achieve the nonlocal effects of subsurface scattering using only locally stored information and a custom shading model. What this means is that our skin shader becomes just that: a simple pixel shader. No extra passes are required and no blurring is required, in texture space nor screen space. Therefore, the cost of our algorithm scales directly with the number of pixels shaded, just like simple shading models such as Blinn-Phong, and it can be implemented on any hardware, with minimal programmable shading support.

1.2 Background and Previous Work

Several offline and real-time approaches have been based on an approach, taken from film, called *texture-space diffusion* (TSD). TSD stores incoming light in texture space and uses a blurring step to simulate diffusion. The first use of this technique was by [Borshukov and Lewis 03,Borshukov and Lewis 05] in the *Matrix* sequels. They rendered light into a texture-space map and then used a custom blur kernel to gather scattered light from all directions. Based on extensive reference to real skin, they used different blur kernels for the red, green, and blue color channels, since different wavelengths of light scatter differently through skin. Since the texture-space diffusion approach used texture-blur operations, it was a very good fit for graphics hardware and was adopted for use in real-time rendering [Green 04, Gosselin et al. 04]. While TSD approaches achieved much more realistic results, the simple blurring operations performed in real time couldn't initially achieve the same level of quality of the expensive, original, nonseparable blurs used in film.

A concept that accurately describes how light diffuses in skin and other translucent materials is known as the *diffusion profile*. For a highly scattering translucent material it is assumed that light scatters equally in all directions as soon as it hits the surface. A diffusion profile can be thought of as a simple plot of how much of this diffused light exits the surface as a function of the distance from the point of entry. Diffusion profiles can be calculated using measured scattering parameters via mathematical models known as *dipole* [Jensen et al. 01] or *multipole* [Donner and Jensen 05] diffusion models. The dipole model works for simpler materials, while the multipole model can simulate the effect of several layers, each with different scattering parameters.

The work by [d'Eon and Luebke 07] sets the current high bar in real-time skin rendering, combining the concept of fast Gaussian texture-space diffusion with the rigor of physically based diffusion profiles. Their approach uses a sum of Gaussians to approximate a multipole diffusion profile for skin, allowing a very large diffusion profile to be simulated using several separable Gaussian blurs. More recent approaches have achieved marked performance improvements. For example, [Hable et al. 09] have presented an optimized texture-space blur kernel, while [Jimenez et al. 09] have applied the technique in screen space.

1.3 Pre-Integrating the Effects of Scattering

We have taken a different approach to the problem of subsurface scattering in skin and have departed from texture-space diffusion (see Figure 1.1). Instead, we wished to see how far we could push realistic skin rendering while maintaining the benefits of a local shading model. Local shading models have the advantage of not requiring additional rendering passes for each object, and scale linearly with the number of pixels shaded. Therefore, rather than trying to achieve subsur-

Figure 1.1. Our pre-integrated skin-shading approach uses the same diffusion profiles as texture-space diffusion, but uses a local shading model. Note how light bleeds over lighting boundaries and into shadows. (Mesh and textures courtesy of XYZRGB.)

face scattering by gathering incoming light from nearby locations (performing an integration during runtime), we instead seek to pre-integrate the effects of subsurface scattering in skin. Pre-integration is used in many domains and simply refers to integrating a function in advance, such that calculations that rely on the function's integral can be accelerated later. Image convolution and blurring are just a form of numerical integration.

The obvious caveat of pre-integration is that in order to pre-integrate a function, we need to know that it won't change in the future. Since the incident light on skin can conceivably be almost arbitrary, it seems as though precomputing this effect will prove difficult, especially for changing surfaces. However, by focusing only on skin rather than arbitrary materials, and choosing specifically where and what to pre-integrate, we found what we believe is a happy medium. In total, we pre-integrate the effect of scattering in three special steps: on the lighting model, on small surface details, and on occluded light (shadows). By applying all of these in tandem, we achieve similar results to texture-space diffusion approaches in a completely local pixel shader, with few additional constraints.

To understand the reasoning behind our approach, it first helps to picture a completely flat piece of skin under uniform directional light. In this particular case, no visible scattering will occur because the incident light is the same everywhere. The only three things that introduce visible scattering are changes in

the surrounding mesh curvature, bumps in the normal map, and occluded light (shadows). We deal with each of these phenomena separately.

1.4 Scattering and Diffuse Light

If we take our previously flat surface with no visible scattering and start to make it a smooth and curvy surface, like skin (we will keep it smooth for now), scattering will start to become visible. This occurs due to the changes in incident light across the surface. The Lambert diffuse-lighting model assumes that diffuse light scatters equally in all directions, and the amount of incident light is proportional to the cosine of the angle between the surface normal and light direction ($N \cdot L$).

Since $N \cdot L$ falloff is a primary cause of changing incident light, and thus visible scattering, there have been several rendering tricks that attempt to add the look of scattering by altering the $N \cdot L$ fall-off itself. This involves making the falloff wrap around the back of objects, or by letting each wavelength of light (r, g, and b) fall off differently as $N \cdot L$ approaches zero. What we found to be the big problem with such approaches is that they aren't based on physical measurements of real skin-like diffusion profiles; and if you tune them to look nice for highly curved surfaces, then there will be a massive falloff for almost-flat surfaces (and vice versa).

To address both issues, we precompute the effect of diffuse light scattering. We do this in a fashion similar to measured *bidirectional reflectance distribution functions* (BRDFs). Measured BRDFs use physically measured data from real surfaces to map incoming to outgoing light. This is as opposed to analytical BRDFs such as Blinn-Phong that are analytical approximations for an assumed micro-facet structure. Typical measured BRDFs don't incorporate $N \cdot L$ since $N \cdot L$ just represents the amount of incoming light and isn't part of the surface's reflectance. We are concerned *only* with $N \cdot L$ (diffuse light), as this is the light that contributes to subsurface scattering.

One approach we considered to precompute the effect of scattered light at any point on a surface, was to simulate lighting from all directions and compress that data using spherical harmonics. Unfortunately, spherical harmonics can efficiently represent only very low frequency changes or would require too many coefficients. Thus, instead of precomputing the effect of scattering at all locations, we chose to precompute the scattering falloff for a subset of surface shapes and determine the best falloff during forward rendering. As discussed above, one characteristic that we can calculate in a shader is surface curvature, which largely determines the effect of scattering on smooth surfaces.

To measure the effect of surface curvature on scattering, we add curvature as the second parameter to our measured diffuse falloff. The skin diffusion profiles from [d'Eon and Luebke 07] on flat skin can be used to simulate the effect of scattering on different curvatures. We simply light a spherical surface of a

given curvature from one direction and measure the accumulated light at each angle with respect to the light (see Figures 1.2 and 1.3). This results in a two-dimensional lookup texture that we can use at runtime. More formally, for each skin curvature and for all angles θ between N and L, we perform the integration in Equation (1.1):

$$D(\theta, r) = \frac{\int_{-\pi}^{\pi} \cos(\theta + x) \cdot R(2r \sin(x/2))dx}{\int_{-\pi}^{\pi} R(2 \sin(x/2))dx} \tag{1.1}$$

The first thing to note is that we have approximated a spherical integration with integration on a ring. We found the difference was negligible and the ring integration fits nicely into a shader that is used to compute the lookup texture. The variable $R()$ refers to the diffusion profile, for which we used the sum of Gaussians from [d'Eon and Luebke 07] (see Table 1.1). Rather than performing an expensive arccos() operation in our shader to calculate the angle, we push this into the lookup, so our lookup is indexed by $N \cdot L$ directly. This is fine in our case, as the area where scattering occurs has plenty of space in the lookup. Figures 1.2 and 1.3 illustrate how to compute and use the diffuse lookup texture.

While this measured skin model can provide some interesting results on its own, it still has a few major flaws. Primarily, it assumes that all skin resembles a sphere, when, in fact, skin can have fairly arbitrary topology. Stated another way, it assumes that scattered light arriving at a given point depends on the curvature of that point itself. In actuality it depends on the curvature of all of the surrounding points on the surface. Thus, this approximation will work very well on smooth surfaces without fast changes in curvature, but breaks down when curvature changes too quickly. Thankfully, most models of skin are broken up into two detail levels: smooth surfaces represented using geometry, and surface

Variance	Red	Green	Blue
0.0064	0.233	0.455	0.649
0.0484	0.100	0.366	0.344
0.187	0.118	0.198	0.0
0.567	0.113	0.007	0.007
1.99	0.358	0.004	0.0
7.41	0.078	0	0.0

Table 1.1. The weights used by [d'Eon and Luebke 07] for texture-space diffusion. Although we aren't limited to the sum of Gaussians approximations, we use the same profile for comparison.

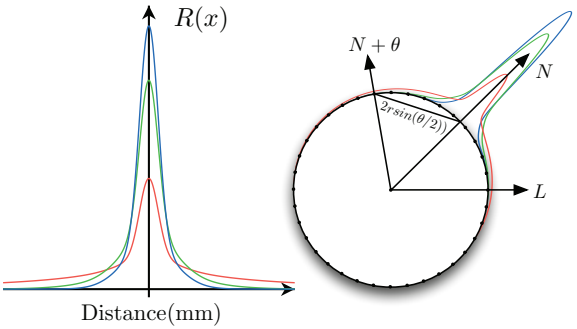


Figure 1.2. The graph (left) illustrates the diffusion profile of red, green, and blue light in skin, using the sum of Gaussians from Table 1.1. The diagram (right) illustrates how we pre-integrate the effect of scattering into a diffuse BRDF lookup. The diffusion profile for skin (overlaid radially for one angle) is used to blur a simple diffuse BRDF for all curvatures of skin.

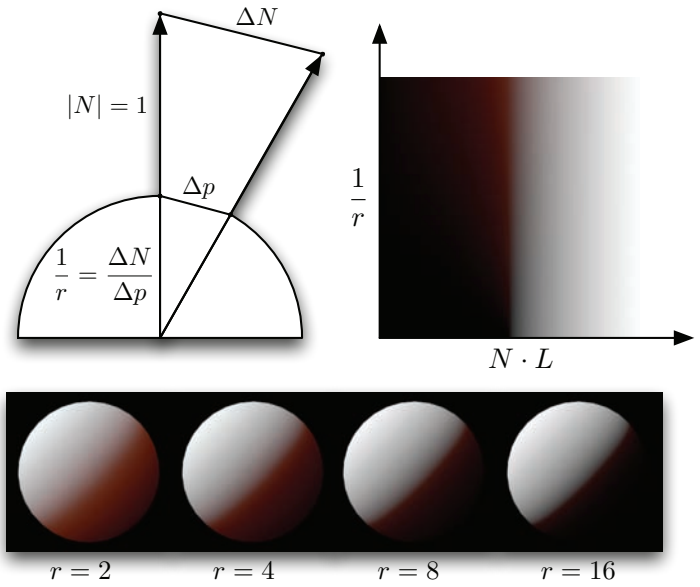


Figure 1.3. The diagram (top left) illustrates how we calculate curvature while rendering using two derivatives. The diffuse BRDF lookup, indexed by curvature (sphere radius) and $N \cdot L$ (top right). Spheres of different sized renderings using the new BRDF lookup (bottom).

details represented in a normal map. We take advantage of this and let the measured diffuse falloff be chosen at the smooth geometry level, while adding another approach to deal with creases and small bumps in normal maps, which are responsible for quick changes in curvature.

1.5 Scattering and Normal Maps

We now turn to the effect of scattering on small wrinkles and pores that are usually represented with a normal map. Since the normal from a small crease always returns to the dominant surface normal, the reflected scattered light coming from that crease will look very similar to light reflected from a nonscattering surface with a physically broader (or blurred-out) crease. Coincidentally, one way of approximating the look of scattered-over small creases and bumps is to simply blur the creases and bumps themselves! Most important however, this effect will be different for each wavelength of light, because of their different diffusion profiles.

Interestingly, the inverse of this phenomenon was noted when capturing normals from a real subject, using image-based lighting. Ma et al. [Ma et al. 07] noted that, when captured using spherical gradient lighting, normals become bent toward the dominant-surface normal, depending on the wavelength of light used to capture them (red was more bent than green, etc.). They also noted that a local skin-shading model was improved by using all the normals they captured instead of only one. In this case the image-based normal capture was physically integrating all the scattered light when determining the best normal to fit the data. Since we have only one set of normal maps to begin with, we essentially work in reverse. We assume that our original normal map is the accurate surface normal map and blur it several times for each wavelength, resulting in a separate normal map for each color, and for specular reflection. As mentioned by previous authors [d'Eon and Luebke 07, Hable et al. 09, Jimenez et al. 09], care should be taken to make sure the original normal map has not been blurred already in an attempt to get a smoother look.

While it might seem to make sense to simply blur the normal map using the diffusion profile of skin, this approach is not completely valid since lighting is not a linear process with regard to the surface normal. What we really want to have is a representation of the normal which can be linearly filtered, much in the same way that shadow maps can be filtered linearly using a technique like variance shadow mapping. Interestingly, there has been some very recent work in linear normal map filtering. *Linear efficient antialiased normal* (LEAN) mapping [Olano and Baker 10] represents the first and second moments of bumps in surface-tangent space. Olano and Baker focused primarily on representing specular light but also suggest a diffuse-filtering approximation from [Kilgard 00] which simply uses the linearly filtered unnormalized normal and standard diffuse lighting. It is noteworthy that the unnormalized normal is actually a valid approximation when

a bump self-shadowing and incident/scattered lighting term is constant over the normal-map filtering region. In that case,

$$\frac{1}{n}\Sigma_{i=1}^{n}(K_{\text{diffuse}}L \cdot N_i) = K_{\text{diffuse}}L \cdot (\frac{1}{n}\Sigma_{i=1}^{n}N_i).$$

The reason this isn't always the case is that diffuse lighting incorporates a self-shadowing term $\max(0, N \cdot L)$ instead of simply $N \cdot L$. This means back-facing bumps will actually contribute negative light when linearly filtered. Nonetheless, using the unnormalized normal will still be valid when all bumps are unshadowed or completely shadowed, and provides a better approximation than the normalized normal in all situations, according to [Kilgard 00].

Although we would prefer a completely robust method of pre-integrating normal maps that supports even changes in incident/scattered light over the filtering region, we found that blurring, using diffusion profiles, provided surprisingly good results (whether or not we renormalize). In addition, since using four normals would require four transformations into tangent space and four times the memory, we investigated an approximation using only one mipmapped normal map. When using this optimization, we sample the specular normal as usual, but also sample a red normal clamped below a tunable miplevel in another sampler. We then transform those two normals into tangent space and blend between them to get green and blue normals. The resulting diffuse-lighting calculations must then be performed three times instead of once. The geometry normal can even be used in place of the second normal map sample, if the normal map contains small details exclusively. If larger curves are present, blue/green artifacts will appear where the normal map and geometry normal deviate, thus the second mipmapped sample is required.

We found that this approach to handling normal maps complements our custom diffuse falloff very well. Since the red normal becomes more heavily blurred, the surface represented by the blurred normal becomes much more smooth, which is the primary assumption made in our custom diffuse falloff. Unfortunately, there is one caveat to using these two approaches together. Since we have separate normals for each color, we need to perform three diffuse lookups resulting in three texture fetches per light. We discuss a few approaches to optimizing this in Section 1.7.

1.6 Shadow Scattering

Although we can now represent scattering due to small- and large-scale features, we are still missing scattering over occluded light boundaries (shadows). The effect of light scattering into shadows is one of the most noticeable features of realistically rendered skin. One would think that scattering from shadows is much more difficult since they are inherently nonlocal to the surface. However, by using

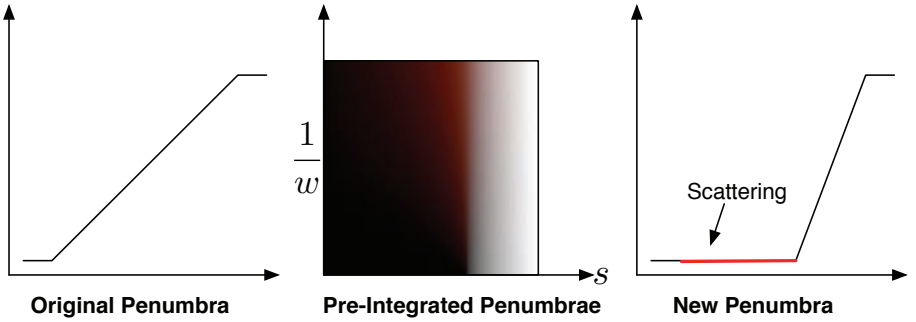


Figure 1.4. Illustration of pre-integrated scattering falloff from shadows. A typical shadow falloff from a box-filtered shadow map (left). A smaller penumbra that we pre-integrate against the diffusion profile of skin (right). The lookup maps the first penumbra into the second but also stores additional scattered light. The lookup is parameterized by the original shadow value and the width of the penumbra in world space (center).

a small trick, we found we could pre-integrate the effect of scattering over shadow boundaries in the same way we represent scattering in our lighting model.

The trick we use for shadows is to think of the results of our shadow mapping algorithm as a falloff function rather than directly as a penumbra. When the falloff is completely black or white, we know we are completely occluded or unoccluded, respectively. However, we can choose to reinterpret what happens between those two values. Specifically, if we ensure the penumbra size created by our shadow map filter is of adequate width to contain most of the diffusion profile, we can choose a different (smaller) size for the penumbra and use the rest of the falloff to represent scattering according to the diffusion profile (see Figure 1.4).

To calculate an accurate falloff, we begin by using the knowledge of the shape of our shadow mapping blur kernel to pre-integrate a representative shadow penumbra against the diffusion profile for skin. We define the representative shadow penumbra $P()$ as a one-dimensional falloff from filtering a straight shadow edge (a step function) against the shadow mapping blur kernel. Assuming a monotonically decreasing shadow mapping kernel, the representative shadow falloff is also a monotonically decreasing function and is thus *invertible* within the penumbra. Thus, for a given shadow value we can find the position within the representative penumbra using the inverse $P^{-1}()$. As an example, for the simple case of a box filter, the shadow will be a linear ramp, for which the inverse is also a linear ramp. More complicated filters have more complicated inverses and need to be derived by hand or by using software like Mathematica. Using the inverse, we can create a lookup texture that maps the original falloff back to its location

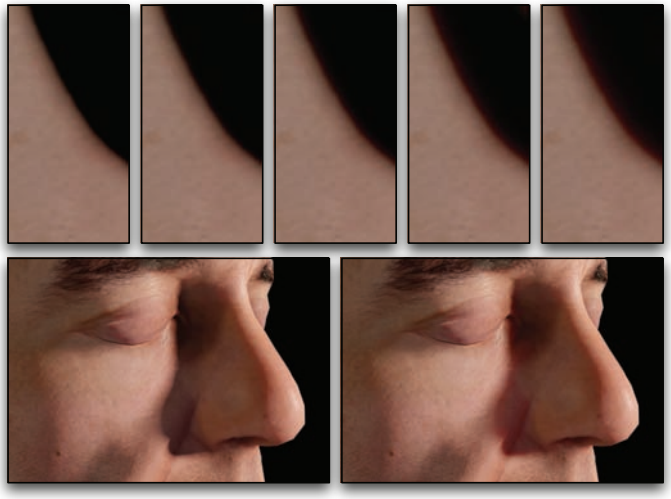


Figure 1.5. Illustration of pre-integrated scattering falloff from shadows. Controlled scattering based on increased penumbra width, such as a penumbra cast onto a highly slanted surface (top). Comparison with and without shadow scattering (bottom).

in the penumbra and then to a completely different falloff. Specifically, we can make the new shadow falloff smaller and use the remainder to represent subsurface scattering from the new penumbra. In the end, we are left with a simple integration to perform that we can use as a lookup during rendering, exactly like our diffuse falloff.

We should note at this point that we could run into problems if our assumptions from above are invalidated. We found that a two-dimensional shadow falloff was not noticeably different from a one-dimensional one, but we have also assumed that all shadow transitions are sharp. For example, if something like a screen door were to cast a shadow, it might result in a constant penumbra value between zero and one. In that case, we would assume there is scattering from a falloff that isn't there. Additionally, we have assumed projection onto a flat surface. If the surface is highly slanted, then the true penumbra size will be much larger than the one we used during pre-integration. For this reason we add a second dimension to our shadow falloff texture, which represents the size of the penumbra in world space. This is similar to the way we pre-integrate lighting against all sphere sizes. In the end, our two-dimensional shadow-lookup integration is a simple convolution:

$$P_S(s,w) = \frac{\int_{-\infty}^{\infty} P'(P^{-1}(s)+x)R(x/w)dx}{\int_{-\infty}^{\infty} R(x/w)dx},$$

where $P^{-1}()$ is the inverse of our representative falloff, $P'()$ is the new, smaller penumbra, $R()$ is the diffusion profile, and s and w are the shadow value and penumbra width in world space, respectively. Penumbra width can be detected using either the angle of the surface with respect to the light, or potentially the derivative of the shadow value itself. Since creating a large penumbra is expensive using conventional shadow filtering (although, see the *Pixel Quad Amortization* chapter), having the penumbra stretched over a slanted surface provides a larger space for scattering falloff and thus can actually be desirable if the initial shadow penumbra isn't wide enough. In this case the lookup can be clamped to insure that the falloff fits into the space provided.

1.7 Conclusion and Future Work

We have presented a new local skin-shading approach based on pre-integration that approximates the same effects found in more expensive TSD-based approaches. Our approach can be implemented by adding our custom diffuse- and shadow-falloff textures to a typical skin shader (see Figure 1.6).

Although we found that our approach worked on a large variety of models, there are still a few drawbacks that should be mentioned. When approximating curvature using pixel shader derivatives, triangle edges may become visible where curvature changes quickly. Depending on how the model was created we also found that curvature could change rapidly or unnaturally in some cases. We are looking into better approaches to approximating curvature in these cases. This is much more easily done with geometry shaders that can utilize surface topology.

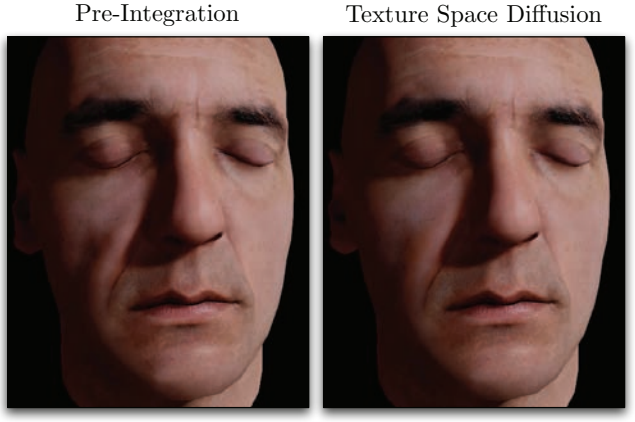


Figure 1.6. Comparison of our approach with texture-space diffusion using an optimized blur kernel from [Hable et al. 09]. (Mesh and textures courtesy of XYZRGB.)

We would also like to look at the effect of using more than one principal axis of curvature. For models where curvature discontinuities occur, we generate a curvature map that can be blurred and further edited by hand, similar to a stretch map in TSD.

Another challenge we would like to meet is to efficiently combine our normal map and diffuse-lighting approaches. When using three diffuse normals, we currently need three diffuse-texture lookups. We found we could use fewer lookups depending on the number of lights and the importance of each light. We have also found it promising to approximate the diffuse and shadow falloffs using analytical approximations that can be evaluated without texture lookups.

We would also like to apply our technique to environment mapping. It should be straightforward to support diffuse-environment mapping via an array of diffuse-environment maps that are blurred based on curvature, in the same manner as our diffuse-falloff texture.

1.8 Appendix A: Lookup Textures

```
float  Gaussian (float  v, float r)
{
    return 1.0/sqrt(2.0*PI*v) * exp(-(r*r)/(2*v));
}

float3 Scatter(float r)
{
   //Coefficients from GPU Gems 3 - ''Advanced Skin Rendering
       .''
   return Gaussian(0.0064 * 1.414,r) * float3
       (0.233,0.455,0.649) +
          Gaussian(0.0484 * 1.414,r) * float3
              (0.100,0.336,0.344) +
          Gaussian(0.1870 * 1.414,r) * float3
              (0.118,0.198,0.000) +
          Gaussian(0.5670 * 1.414,r) * float3
              (0.113,0.007,0.007) +
          Gaussian(1.9900 * 1.414,r) * float3
              (0.358,0.004,0.000) +
          Gaussian(7.4100 * 1.414,r) * float3
              (0.078,0.000,0.000);
}

float3 integrateShadowScattering(float penumbraLocation,
                                 float penumbraWidth)
{
    float3 totalWeights = 0;
    float3 totalLight = 0;

    float a= -PROFILE_WIDTH;
```

```
    while( a<=PROFILE_WIDTH )
    while( a<=PROFILE_WIDTH )
    {
        float light = newPenumbra(penumbraLocation + a/
            penumbraWidth);
        float sampleDist = abs(a);
        float3 weights = Scatter(sampleDist);
        totalWeights += weights;
        totalLight += light * weights;
        a+=inc;
    }

    return totalLight / totalWeights;
}

float3 integrateDiffuseScatteringOnRing(float cosTheta,float
    skinRadius)
{
    // Angle from lighting direction.
    float  theta = acos(cosTheta);
    float3 totalWeights = 0;
    float3 totalLight = 0;

    float a= -(PI/2);
    while( a<=(PI/2) )
    while( a<=(PI/2) )
    {
        float sampleAngle = theta + a;
        float diffuse = saturate( cos(sampleAngle) );
        float sampleDist = abs(2.0*skinRadius*sin(a*0.5));
            // Distance.
        float3 weights = Scatter(sampleDist);
            // Profile Weight.
        totalWeights += weights;
        totalLight += diffuse * weights;
        a+=inc;
    }
    return totalLight / totalWeights;
}
```

Listing 1.1. Shader code to precompute skin falloff textures.

1.9 Appendix B: Simplified Skin Shader

```
float3 SkinDiffuse( float curv, float3 NdotL )
{
    float3 lookup = NdotL * 0.5 + 0.5;
    float3 diffuse;
```

```
    diffuse.r = tex2D(SkinDiffuseSampler, float2(lookup.r, curv
        )).r;
    diffuse.g = tex2D(SkinDiffuseSampler, float2(lookup.g, curv
        )).g;
    diffuse.b = tex2D(SkinDiffuseSampler, float2(lookup.b, curv
        )).b;
    return diffuse;
}

float3 SkinShadow( float shad, float width )
{
    return tex2D(SkinShadowSampler, float2(shad, width)).rgb;
}
...
//Simple curvature calculation.
float curvature = saturate(length(fwidth(Normal)) /
                  length(fwidth(WorldPos)) * tuneCurvature ) ;
...
//Specular/Diffuse Normals.
float4 normMapHigh   = tex2D(NormalSamplerHigh, Uv) * 2.0 -
    1.0;
float4 normMapLow    = tex2D(NormalSamplerLow , Uv) * 2.0 -
    1.0;
float3 N_high = mul(normMapHigh.xyz, TangentToWorld);
float3 N_low  = mul(normMapLow.xyz, TangentToWorld);
float3 rS = N_high;
float3 rN = lerp(N_high, N_low, tuneNormalBlur.r);
float3 gN = lerp(N_high, N_low, tuneNormalBlur.g);
float3 bN = lerp(N_high, N_low, tuneNormalBlur.b);
...
//Diffuse lighting
float3 NdotL = float3( dot(rN,L), dot(gN,L), dot(bN,L) );
float3 diffuse = SkinDiffuse( curvature, NdotL ) * LightColor *
                 SkinShadow( SampleShadowMap(ShadowUV) );
```

Listing 1.2. Skin shader example.

Bibliography

[Borshukov and Lewis 03] George Borshukov and J.P. Lewis. "Realistic Human Face Rendering for The Matrix Reloaded." In *ACM Siggraph Sketches and Applications*. New York: ACM, 2003.

[Borshukov and Lewis 05] George Borshukov and J.P. Lewis. "Fast Subsurface Scattering." In *ACM Siggraph Course on Digital Face Cloning*. New York: ACM, 2005.

[d'Eon and Luebke 07] E. d'Eon and D. Luebke. "Advanced Techniques for Realistic Real-Time Skin Rendering." In *GPU Gems 3*, Chapter 14. Reading, MA: Addison Wesley, 2007.

[Donner and Jensen 05] Craig Donner and Henrik Wann Jensen. "Light Diffusion in Multi-Layered Translucent Materials." *ACM Trans. Graph.* 24 (2005), 1032–1039.

[Gosselin et al. 04] D. Gosselin, P.V. Sander, and J.L. Mitchell. "Real-Time Texture-Space Skin Rendering." In *ShaderX*^{3}*: Advanced Rendering with DirectX and OpenGl*. Hingham, MA: Charles River Media, 2004.

[Green 04] Simon Green. "Real-Time Approximations to Subsurface Scattering." In *GPU Gems*, pp. 263–278. Reading, MA: Addison-Wesley, 2004.

[Hable et al. 09] John Hable, George Borshukov, and Jim Hejl. "Fast Skin Shading." In *ShaderX*^{7}*: Advanced Rendering Techniques*, Chapter II.4. Hingham, MA: Charles River Media, 2009.

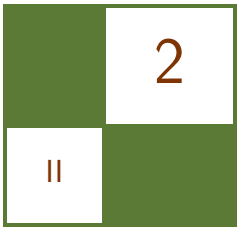
[Jensen et al. 01] Henrik Jensen, Stephen Marschner, Mark Levoy, and Pat Hanrahan. "A Practical Model for Subsurface Light Transport." In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, *SIGGRAPH '01*, pp. 511–518. New York: ACM, 2001.

[Jimenez et al. 09] Jorge Jimenez, Veronica Sundstedt, and Diego Gutierrez. "Screen-Space Perceptual Rendering of Human Skin." *ACM Transactions on Applied Perception* 6:4 (2009), 23:1–23:15.

[Kilgard 00] Mark J. Kilgard. "A Practical and Robust Bump-Mapping Technique for Today's GPUs." In *GDC 2000*, 2000.

[Ma et al. 07] W.C. Ma, T. Hawkins, P. Peers, C.F. Chabert, M. Weiss, and P. Debevec. "Rapid Acquisition of Specular and Diffuse Normal Maps from Polarized Spherical Gradient Illumination." In *Eurographics Symposium on Rendering*. Aire-la-Ville, Switzerland: Eurographics Association, 2007.

[Olano and Baker 10] Marc Olano and Dan Baker. "LEAN mapping." In *ACM Siggraph Symposium on Interactive 3D Graphics and Games*, pp. 181–188. New York: ACM, 2010.



Implementing Fur Using Deferred Shading

Donald Revie

This chapter is concerned with implementing a visually pleasing approximation of fur using deferred shading rather than attempting to create an accurate physical simulation. The techniques presented can also be used to create a number of materials that are traditionally difficult to render in deferred shading.

2.1 Deferred Rendering

For the purposes of this chapter, the term *deferred rendering* can be extended to any one of a group of techniques characterized by the separation of lighting calculations from the rendering of light-receiving objects within the scene, including deferred shading [Valient 07], deferred lighting [Mittring 09], inferred lighting [Kircher 09], and light-prepass rendering [Engel 09]. The fur-rendering technique being presented has been implemented in deferred shading but should be applicable to any rendering solution based on one of these techniques.

This separation of light-receiving objects from light sources is achieved by storing all relevant information about light-receiving objects in the scene as texture data, collectively referred to as a *geometry buffer* or G-buffer because it represents the geometric scene.

When rendering the lights, we can treat the G-buffer as a screen-aligned quad with per-pixel lighting information. Rendering the G-buffer discards all occluded geometry, effectively reducing the three-dimensional scene into a continuous screen-facing surface (Figure 2.1). By using the two-dimensional screen position, depth, and normal information, a pixel shader can reconstruct any visible point in the scene from its corresponding pixel. It is this surface information that is used to calculate lighting values per pixel rather than the original scene geometry.

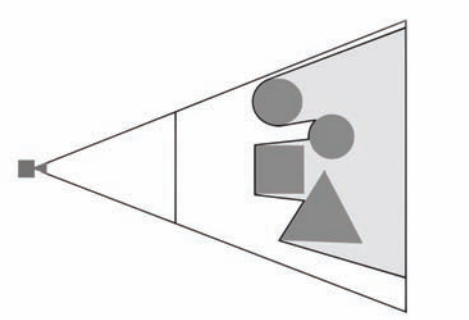


Figure 2.1. G-buffer surface.

In deferred rendering the format of the G-buffer (Figure 2.2) defines a standard interface between all light-receiving materials and all light sources. Each object assigned a light-receiving material writes a uniform set of data into the G-buffer, which is then interpreted by each light source with no direct information regarding the original object. One key advantage to maintaining this interface is that geometric complexity is decoupled from lighting complexity.

This creates a defined pipeline (Figure 2.3) in which we render all geometry to the G-buffer, removing the connection between the geometric data and individual objects, unless we store this information in the G-buffer. We then calculate lighting from all sources in the scene using this information, creating a light-accumulation buffer that again discards all information about individual lights. We can revisit this information in a material pass, rendering individual meshes again and using the screen-space coordinates to identify the area of the light-accumulation buffer and G-buffer representing a specific object. This material phase is required in deferred lighting, inferred lighting, and light pre-pass rendering to complete the lighting process since the G-buffer for these techniques

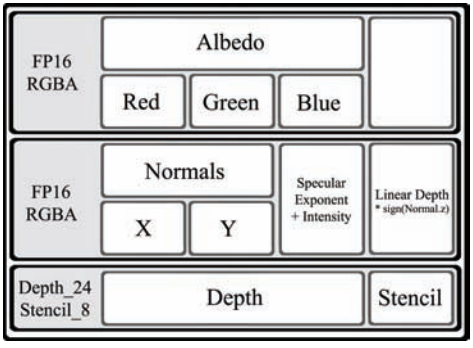


Figure 2.2. Our G-buffer format.

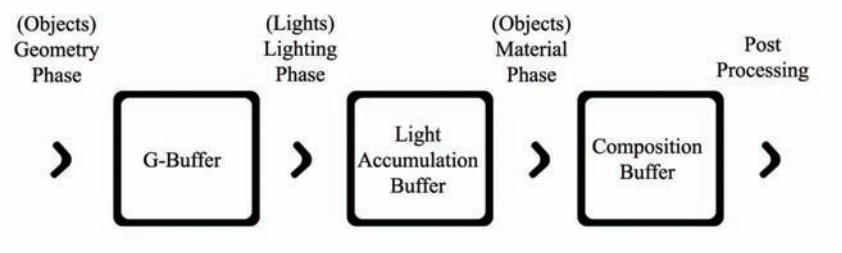


Figure 2.3. General deferred rendering pipeline.

does not include surface color. After this, a post-processing phase acts upon the contents of the composition buffer, again without direct knowledge of individual lights or objects.

This stratification of the deferred rendering pipeline allows for easy extensibility in the combination of different materials and lights. However, adherence to the interfaces involved also imposes strict limitations on the types of materials and lights that can be represented. In particular, deferred rendering solutions have difficulty representing transparent materials, because information regarding surfaces seen through the material would be discarded. Solutions may also struggle with materials that reflect light in a nontypical manner, potentially increasing the complexity of all lighting calculations and the amount of information required within the G-buffer. Choosing the right phases and buffer formats are key to maximizing the power of deferred rendering solutions.

We describe techniques that address the limitations of rendering such materials while continuing to respect the interfaces imposed by deferred rendering. To illustrate these techniques and demonstrate ways in which they might be combined to form complex materials, we outline in detail a solution for implementing fur in deferred shading.

2.2 Fur

Fur has a number of characteristics that make it difficult to represent using the same information format commonly used to represent geometry in deferred rendering solutions.

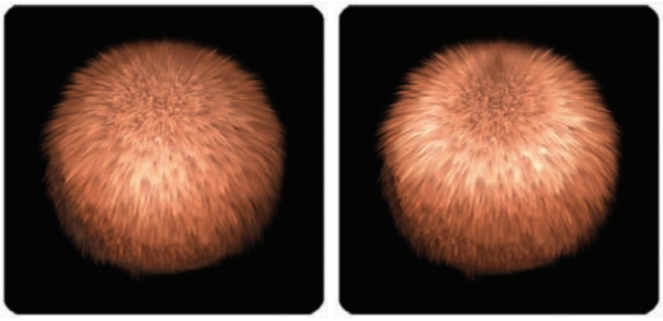
Fur is a structured material composed of many fine strands forming a complex volume rather than a single continuous surface. This structure is far too fine to describe each strand within the G-buffer on current hardware; the resolution required would be prohibitive in both video memory and fragment processing. As this volumetric information cannot be stored in the G-buffer, the fur must be approximated as a continuous surface when receiving light. We achieve this by ensuring that the surface exhibits the observed lighting properties that would normally be created by the structure.

(a) (b)

Figure 2.4. Fur receiving light from behind (a) without scattering and (b) with scattering approximation.

The diffuse nature of fur causes subsurface scattering; light passing into the volume of the fur is reflected and partially absorbed before leaving the medium at a different point. Individual strands are also transparent, allowing light to pass through them. This is often seen as a halo effect; fur is silhouetted against a light source that illuminates the fur layer from within, effectively bending light around the horizon of the surface toward the viewer. This is best seen in fur with a loose, "fluffy" structure (see Figure 2.4).

The often-uniform, directional nature of fur in combination with the structure of individual strands creates a natural grain to the surface being lit. The reflectance properties of the surface are anisotropic, dependent on the grain direction. Anisotropy occurs on surfaces characterized by fine ridges following the grain of the surface, such as brushed metal, and causes light to reflect according to the direction of the grain. This anisotropy is most apparent in fur that is "sleek" with a strong direction and a relatively unbroken surface (see Figure 2.5).



(a) (b)

Figure 2.5. Fur receiving light (a) without anisotropy and (b) with anisotropy approximation.

2.3 Techniques

We look at each of the characteristics of fur separately so that the solutions discussed can be reused to represent other materials that share these characteristics and difficulties when implemented within deferred rendering.

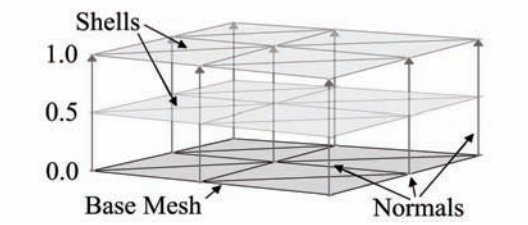


Figure 2.6. Concentric shells.

2.3.1 Volumetric Fur Rendering Using Concentric Shells

It is common to render volumetric structures in real time by rendering discrete slices of volumetric texture data into the scene and using alpha blending to combine the results, such as light interacting with dust particles in the air [Mitchell 04]. Provided enough slices are rendered, the cumulative result gives the appearance of a continuous volume featuring correct perspective, parallax, and occlusion.

The concentric shells method of rendering fur [Lengyel 01] represents the volumetric layer of fur as a series of concentric shells around the base mesh; each shell is a slice through the layer of fur parallel to the surface. These shells are constructed by rendering the base mesh again and pushing the vertices out along the normal of the vertex by a fraction of the fur layer depth; the structure of the fur is represented by a volume texture containing a repeating section of fur (see Figure 2.6). By applying an offset parallel to the mesh surface in addition to the normal we can comb the fur volume (see Figure 2.7, Listing 2.1).

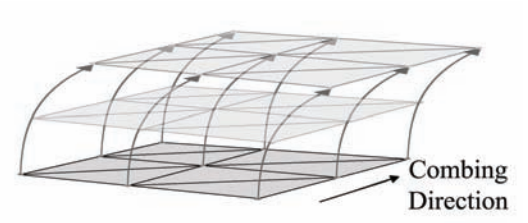


Figure 2.7. Combing.

```
// Get shell depth as normalized distance between base and
// outer surface.
float shellDepth = shellIndex * (1.f/numShells);

// Get offset direction vector
float3 dir = IN.normal.xyz + (IN.direction.xyz * _shellDepth);
dir.xyz = normalize(dir.xyz);

// Offset vertex position along fur direction.
OUT.position = IN.position;
OUT.position.xyz = (dir.xyz * _ shellDepth * furDepth
    * IN.furLength);
OUT.position = mul(worldViewProjection, OUT.position);
```

Listing 2.1. Vertex offsetting.

This method of fur rendering can be further augmented with the addition of *fins*, slices perpendicular to the surface of the mesh, which improve the quality of silhouette edges. However, fin geometry cannot be generated from the base mesh as part of a vertex program and is therefore omitted here (details on generating fin geometry can be found in [Lengyel 01]).

This technique cannot be applied in the geometry phase because the structure of fur is constructed from a large amount of subpixel detail that cannot be stored in the G-buffer where each pixel must contain values for a discrete surface point. Therefore, in deferred shading we must apply the concentric shell method in the material phase, sampling the lighting and color information for each hair from a single point in the light-accumulation buffer. The coordinates for this point can be found by transforming the vertex position of the base mesh into screen space in the same way it was transformed originally in the geometry phase (Listing 2.2).

```
// Vertex shader.
// See (Listing 3.1.1) for omitted content.
// Output screen position of base mesh vertex.
OUT.screenPos = mul(worldViewProjection, IN.position);

// ---------------------------------
// Pixel shader.
IN.screenPos /= IN.screenPos.w;

// Bring values into range (0,1) from (-1,1).
float2 screenCoord = (IN.screenPos.xy + 1.f.xx) * 0.5f.xx;

// Sample lit mesh color
color = tex2D(lightAccumulationTexture, screenCoord).
```

Listing 2.2. Sampling lit objects.

This sampling of lighting values can cause an issue specific to rendering the fur. As fur pixels are offset from the surface being sampled, it is possible for the original point to have been occluded by other geometry and thus be missing from the G-buffer. In this case the occluding geometry, rather than the base mesh, would contribute to the coloring of the fur leading to visual artifacts in the fur (Figure 2.8). We explore a solution to this in Sections 2.3.4 and 2.4.4 of this article.

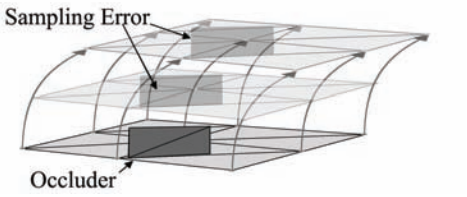


Figure 2.8. Occlusion error.

2.3.2 Subsurface Scattering

Scattering occurs in materials wherein light enters the surface at one point, is transmitted through the medium beneath the surface being reflected and refracted by internal structures and is partially absorbed, before exiting the surface at a different point (Figure 2.9). This light exiting the surface softens the appearance of lighting on the surface by creating a subtle glow.

Much work has been done on the approximation of subsurface scattering properties in skin that is constructed of discrete layers, each with unique reflectance properties. One such solution is to apply a weighted blur to the light accumulated on the surface [Hable 09, Green 04]. In existing forward shaded solutions, this blurring is typically applied in texture space.

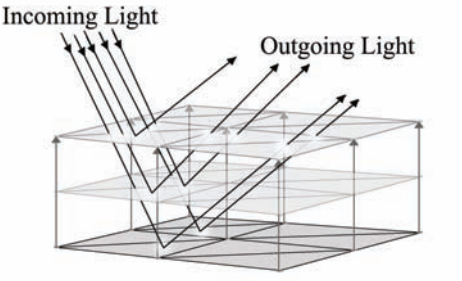


Figure 2.9. Simple subsurface scattering.

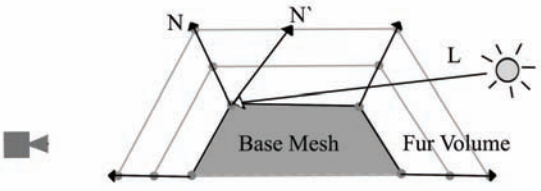


Figure 2.10. Rim glow (N.L < 0) (N'.L > 0).

In deferred rendering, this technique can be applied in both the geometry phase and the material phase. In the geometry phase the scattering can be approximated by blurring the surface normals written into the G-buffer or by recalculating the mesh normals as a weighted sum of neighboring vertex normals [Patro 07].

Blurring can be performed in the material phase, in texture space, by sampling the accumulated lighting in the same manner as that used for the fur rendering. The texture coordinates of the mesh would then be used as vertex positions to write those values into the mesh's texture space before applying a blur. Once blurred, these values are written back into the light-accumulation buffer by reversing the process. Alternatively, the material-phase blurring could be performed in screen space by orienting the blur kernel to the local surface, using the normals stored in the G-buffer at each pixel.

One issue with this solution is that scattering beneath a surface will also allow light entering the back faces of an object to be transmitted through the medium and exit the surface facing the viewer. In materials such as skin and fur, which form a scattering layer over a more solid structure, this transfer of light appears most often around the silhouette edges of the object. We can adjust for this by bending normals at the silhouette edge of the mesh to point away from the viewer and sample lighting from behind the object (see Figure 2.10 and Listing 2.3). In doing so, these pixels will no longer receive direct lighting correctly; this must then be accounted for during the blur phase (see Sections 2.3.4 and 2.4.4 for details of our solution).

```
// Get normal based for back face samples.
// Glow strength and falloff are supplied by material values.

half NdotV = saturate(dot(normal.xyz, -view);
half rimWeight = glowStrenth * pow(1.f - NdotV), glowFalloff);
normal.xyz += view.xyz * rimWeight;
normal.xyz = normalize(normal.xyz);
```

Listing 2.3. Pushing edge pixels around edges.

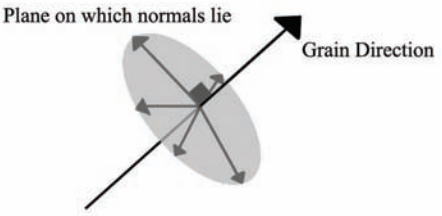


Figure 2.11. Strand normals.

2.3.3 Anisotropy

Anisotropic light reflection occurs on surfaces where the distribution of surface normals is dependent on the surface direction; such surfaces are often characterized by fine ridges running in a uniform direction across the surface, forming a "grain." The individual strands in fur and hair can create a surface that exhibits this type of lighting [Scheuermann 04].

This distinctive lighting is created because in anisotropic surfaces the ridges or, in this case, the strands are considered to be infinitely fine lines running parallel to the grain. These lines do not have a defined surface normal but instead have an infinite number of possible normals radiating out perpendicularly to their direction (see Figure 2.11). Therefore, the lighting calculation at any point on the surface must integrate the lighting for all the normals around the strand. This is not practical in a pixel shader; the best solution is to choose a single normal that best represents the lighting at this point [Wyatt 07].

In forward shading, anisotropy is often implemented using a different lighting calculation from those used to describe other surfaces (Listing 2.4) [Heidrich 98]. This algorithm calculates lighting based on the grain direction of the surface rather than the normal.

```
Diffuse =  sqrt(1 - (< L,T >)^2)
Specular = sqrt(1 - (< L,T >)^2) sqrt(1 - (< V, T > )^2)
    - < L, T >  < V, T >
```

Listing 2.4. Anisotropic light calculation.

In deferred shading we cannot know in the geometry phase the nature of any light sources that might contribute to the lighting on the surface and are bound by the interface of the G-buffer to provide a surface normal. Therefore, we define

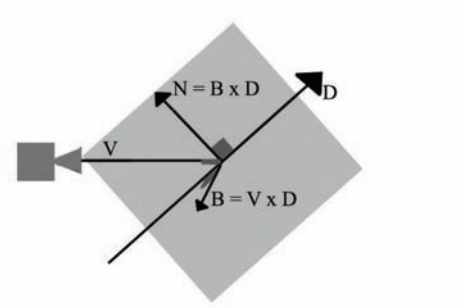
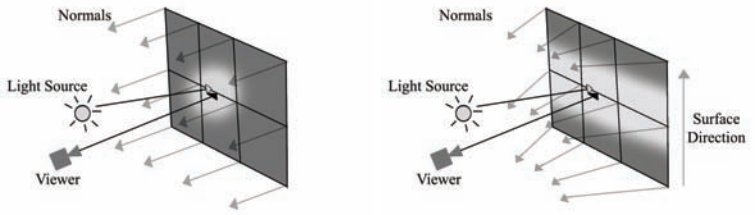


Figure 2.12. Normal as tangent to plane.

the most significant normal as the normal that is coplanar with the grain direction and the eye vector at that point (see Figure 2.12). We calculate the normal of this plane as the cross product of the eye vector and the grain direction, the normal for lighting is then the cross product of the plane's normal and the grain direction (see Listing 2.5).

```
// Generate normal from fur direction.
IN.direction = IN.direction-(dot(IN.direction, normal) * normal);
IN.direction.xyz = normalize(IN.direction.xyz);
half3 binorm = cross(IN.eyeVector, IN.direction);
half3 grainNorm = cross(binorm, IN.direction);
normalize(grainNorm);
```

Listing 2.5. Anisotropic normal calculation.



(a) (b)

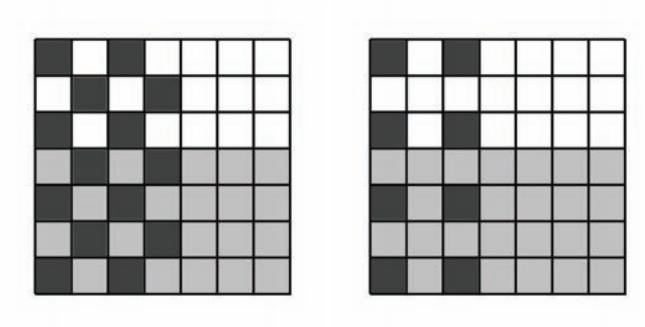
Figure 2.13. (a) Isotropic highlight and (b) anisotropic highlight.

By calculating surface normals in this way we create the effect of curving the surface around the view position, resulting in the lighting being stretched perpendicular to the surface grain (Figure 2.13). While this method does not perfectly emulate the results of the forward-shading solution, it is able to generate

this characteristic of stretched lighting for all light sources, including image-based lights.

2.3.4 Stippled Rendering

Stippled rendering is a technique in which only some pixels of an image are written into the frame buffer, leaving other pixels with their original values. This technique was originally inspired by the stippled alpha transparencies used in games before the widespread availability of hardware alpha blending, also referred to as screen-door transparency [Mulder 98]. The values for the transparent object are written to only some of the pixels covered by the object so as not to completely obscure the scene behind it (see Figure 2.14 and Listing 2.6).



(a) (b)

Figure 2.14. Stipple patterns (a) 1 in 2 and (b) 1 in 4.

```
// Get screen coordinates in range (0, 1).
float2 screenCoord = ((IN.screenPos.xy/IN.screenPos.w)
   + 1.f.xx) * 0.5h.xx;
// Convert coordinates into pixels.
int2 sample = screenCoord.xy * float2(1280.f, 720.f);

// If pixel is not the top left in a 2x2 tile discard it.
int2 tileIndices = int2(sample.x \% 2, sample.y \% 2);
if((tileIndices.x != 0) || (tileIndices.y != 0))
discard;
```

Listing 2.6. One in four Stipple pattern generation.

In deferred shading, transparent objects are written into the G-buffer using a stipple pattern. During the material phase, the values of pixels containing data for the transparent surface are blended with neighboring pixels containing

information on the scene behind. By varying the density of the stipple pattern, different resolutions of data can be interleaved, allowing for multiple layers of transparency. This technique is similar to various interlaced rendering methods for transparent surfaces [Pangerl 09, Kircher 09].

The concept of stippled rendering can be extended further to blend multiple definitions of a single surface together. By rendering the same mesh multiple times but writing distinctly different data in alternating pixels, we can assign multiple lighting values for each point on the object at a reduced resolution. During the material phase the object is rendered again, and this information is deinterlaced and combined to allow for more complex lighting models. For example, a skin material could write separate values for a subsurface scattering pass and a specular layer, as interleaved samples. The material pass would then additively blend the specular values over the blurred result of the diffuse lighting values.

2.4 Fur Implementation Details

Ease of use and speed of implementation were key considerations when developing the fur solution. We found that to enable artists to easily apply the fur material to meshes, it was important to provide flexibility through a fine degree of control, coupled with real-time feedback. We also wished to ensure minimal changes to existing assets and work methods. It was also important that the technique have minimal impact on our rendering framework, and that it work well with our existing asset-creation pipeline.

To this end, the solution has been implemented with minimal code support; all shader code is implemented within a single effect file with multiple passes for the geometry and material phases of rendering. Annotations provide the renderer with information on when and where to render passes. For real-time feedback, a separate technique is provided within the effect file that renders the fur in a forward-shaded fashion suitable for use within various asset-creation packages.

2.4.1 Asset Preparation

Combing direction. The fur solution is applicable to any closed mesh with per-vertex position, normal, tangent, binormal, and a single set of two-dimensional texture coordinates. This is a fairly standard vertex format for most asset-creation packages.

In addition, we require an RGBA color per vertex to define the combing direction and length of fur at a given vertex (see Figure 2.15). The RGB components encode combing direction as a three-component vector in the object's local space compressing a range of $[-1, 1]$ to $[0, 1]$; this vector is also used to describe the surface direction when generating the anisotropic surface normals. The alpha channel of the color is used to scale the global fur length locally at each vertex.

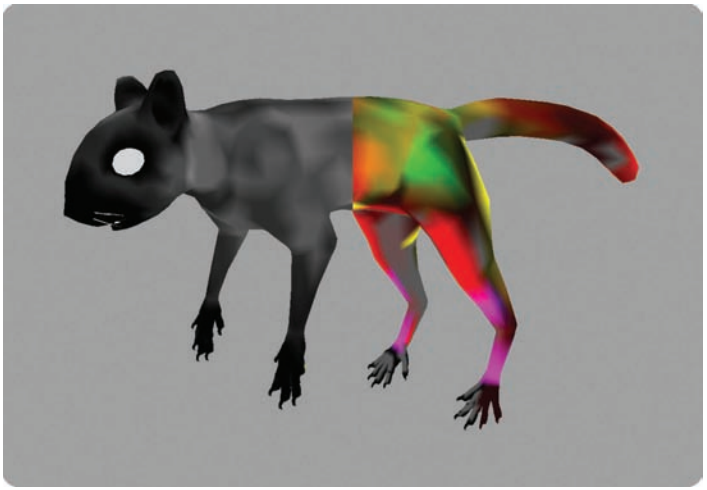


Figure 2.15. Fur length (left) and direction (right) encoded as color.

A color set was chosen to encode this data for a number of reasons. First, many asset-creation tools allow for easy "painting" of vertex colors while viewing the shaded mesh in real time. This effectively allows the author to choose a direction represented as a color and then comb sections of the fur appropriately using the tool, setting an alpha component to the color trims the length of the fur locally. Second, the approach of encoding direction as color is already familiar to most authors through the process of generating world- and tangent-space normal maps. The process has proven to be quite intuitive and easy to use.

As part of the loading process, we transform the vectors encoded in this color channel from the local space of the mesh into its tangent space and at the same time orthonormalize them, making them tangential to the mesh surface. Thus when the mesh is deformed during animation, the combing direction of the fur will remain constant in relation to the surface orientation. This is the only engine side code that was required to fully support the fur-rendering technique (see Listing 2.7).

```
// Build local to tangent space matrix.
Matrix tangentSpace;
tangentSpace.LoadIdentity();
tangentSpace.SetCol(0, tangent);
tangentSpace.SetCol(1, binormal);
tangentSpace.SetCol(2, normal);
tangentSpace.Transpose();

// Convert color into vector.
```

```
Vector3 dir(pColour[0], pColour[1], pColour[2]);
dir = (dir * 2.f) - Vector3(1.f);

// Gram Schmidt orthonormalization.
dir = dir - (dot(dir, normal) * normal); dir.Normalise();

// Transform vector into tangent space.
tangentSpace.TransformInPlace(dir);

// Convert vector into color.
dir = (dir + Vector3(1.f)) * 0.5;
pColour[0] = dir.getX();
pColour[1] = dir.getY();
pColour[2] = dir.getZ();
```

Listing 2.7. Processing of fur directions.

Texture data. To provide the G-buffer with the necessary surface information, the material is assigned an RGB albedo map and a lighting map containing per pixel normal information and specular intensity and exponent at any given pixel. In addition to this, a second albedo map is provided to describe the changes applied to lighting as it passes deeper into the fur; over the length of the strands, the albedo color that is used is blended from this map to the surface color. This gives the author a high degree of control over how the ambient occlusion term is applied to fur across the whole surface, allowing for a greater variation.

To represent the fur volume required for the concentric shell rendering, a heightfield was chosen as an alternative to generating a volumetric data set. While this solution restricts the types of volume that can be described, it requires considerably less texture information to be stored and accessed in order to render the shells. It is more flexible in that it can be sampled using an arbitrary number of slices without the need to composite slices when undersampling the volume, and it is far simpler to generate with general-image authoring tools.

2.4.2 Geometry Phase

The geometry phase is split into two passes for the purpose of this technique. The first pass renders the base mesh to the G-buffer. In the vertex shader the position, tangent, and normal are transformed into view space and the combing direction is brought into local space in the range [-1, 1]. The pixel shader generates a new normal, which is coplanar to the eye and combing vectors to achieve anisotropic highlights (Figure 2.16).

The second pass renders the top layer of the fur in a stipple pattern, rendering to one in every four pixels on screen. The vertex shader is identical to the first pass, but pushes the vertex positions out along the vertex normals offset by the

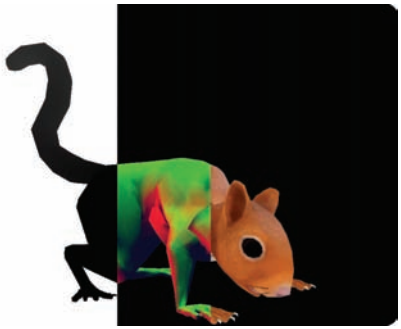


Figure 2.16. Geometry pass 1 (depth/normals/albedo).

global fur length scaled by the vertex color alpha. The pixel shader identifies likely silhouette edges using the dot product of the view vector and the surface normals; the normals at these points are adjusted by adding the view vector scaled by this weight value. The unmodified normals are recalculated to use the anisotropic normals like those of the first pass (Figure 2.17).

This second pass solves the occlusion issue when constructing concentric fur shells from the light-accumulation buffer, since both samples are unlikely to be occluded simultaneously while any part of the strand is still visible. The second pass allows light calculations to be performed for both the surface of the mesh and also the back faces where light entering the reverse faces may be visible.

In order to avoid incorrect results from *screen-space ambient occlusion* (SSAO), edge detection, and similar techniques that rely on discontinuities in the G-buffer, these should be calculated before the second pass since the stipple pattern will create artifacts.

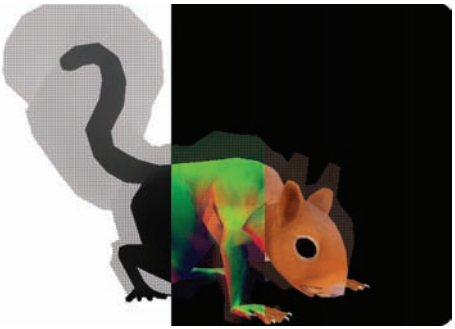


Figure 2.17. Geometry pass 2 (depth/normals/albedo).

2.4.3 Light Phase

During the light phase both the base and stipple samples within the G-buffer receive lighting in the same manner as all other values in the G-buffer, adherence to a common interface allows the fur to receive lighting from a wide range of sources.

Figure 2.18. Light-accumulation buffer.

2.4.4 Material Phase

The material phase of rendering involves reading the values from the light-accumulation buffer and interpreting these based on specific qualities of the material, in this case by constructing shells of fur. In deferred shading, since the majority of the lighting values are already correct in the light-accumulation buffer, a copy of these values is required onto which the material phase of the fur can be composited (see Figure 2.18).

The stipple values, being distributed on the outermost shell of the fur, will occlude the layers of fur beneath. To correct this, all fur surfaces must be rendered again using the outermost shell, while sampling color values from the light-accumulation buffer and depth values from the linear depth stored in the G-buffer (see Figure 2.19). For most pixels, these color and depth values are written directly into the composition buffer, however, where a stipple value would be sampled the neighboring pixel is used instead, effectively erasing all stipple values from the light-accumulation and depth buffers.

The buffer now contains the base mesh of the object only, providing a basis on which to composite the volumetric layer of fur. Rendering of the fur is performed by a series of passes, each pass rendering a concentric shell by offsetting the vertex positions. The pass also constructs positions in screen space, from which both the sample corresponding to the base mesh and the stipple sample corresponding to the outermost shell can be obtained.

Figure 2.19. Stipple obliteration pass.

In the pixel shader these two samples are retrieved from the light-accumulation buffer, their respective linear depths in the G-buffer are also sampled to compare against the depth of the sample coordinates and thus correct for occlusion errors. If both samples are valid, the maximum of the two is chosen to allow for the halo effect of scattering around the edges of the object without darkening edges where there is no back lighting. The contribution of the albedo map to the accumulated light values is removed by division and then reapplied as a linear interpolation of the base and top albedo maps to account for ambient occlusion by the fur. The heightfield for the fur volume is sampled at a high frequency by applying an arbitrary scale to the mesh UVs in the material. The smoothstep function is used to fade out pixels in the current shell as the interpolation factor equals and exceeds the values stored in the heightfield, thus individual strands of fur fade out at different rates, creating the impression of subpixel detail (see Figure 2.20).

Figure 2.20. Shell pass (16 shells).

Figure 2.21. Final image.

2.5 Conclusion

This article has described a series of techniques used to extend the range of materials that can be presented in a deferred rendering environment, particularly a combination of these techniques that can be used to render aesthetically pleasing fur at real-time speeds.

2.6 Acknowledgments

Thanks to everyone at Cohort Studios for showing their support, interest, and enthusiasm during the development of this technique, especially Bruce McNeish and Gordon Bell, without whom there would be no article.

Special thanks to Steve Ions for patiently providing excellent artwork and feedback while this technique was still very much in development, to Baldur Karlsson and Gordon McLean for integrating the fur, helping track down the (often humorous) bugs, and bringing everything to life, and to Shaun Simpson for all the sanity checks.

Bibliography

[Engel 09] W. Engel. "The Light Pre-Pass Renderer." In *ShaderX*^{7}, pp. 655–666. Hingham, MA: Charles River Media, 2009.

[Green 04] S. Green. "Real-Time Approximations to Sub-Surface Scattering." In *GPU Gems*, pp. 263–278. Reading, MA: Addison Wesley, 2004.

[Hable 09] J. Hable, G. Borshakov, and J. Heil. "Fast Skin Shading." In *ShaderX*^{7}, pp. 161–173. Hingham, MA: Charles River Media, 2009.

[Heidrich 98] W. Heidrich and Hans-Peter Seidel. "Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware." In Proceedings of Image and Multi-Dimensional Digital Signal Processing Workshop,Washington, DC: IEEE, 1998.

[Kircher 09] S. Kircher and A Lawrance. "Inferred Lighting: Fast Dynamic Lighting and Shadows for Opaque and Translucent Objects." In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, Sandbox '09, pp. 39–45. New York: ACM, 2009.

[Lengyel 01] J. Lengyel, E. Praun, A. Finkelstein, and H. Hoppe. "Real-Time Fur Over Arbitrary Surfaces." In *I3D '01 Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pp. 227–232. New York, ACM Press, 2001.

[Mitchell 04] J. Mitchell. "Light Shafts: Rendering Shadows in Participating Media." Game Developers Conference, 2004. Availablwe online at http://developer.amd.com/media/gpu_assets/Mitchell_LightShafts.pdf.

[Mittring 09] M. Mittring. "A Bit More Deferred - CryEngine3." Triangle Game Conference, 2009.

[Mulder 98] J. D. Mulder, F. C. A. Groen, and J. J. van Wijk. "Pixel Masks for Screen-Door Transparency." In *Proceedings of the Conference on Visualization '98*, pp. 351–358. Los Alamitos, CA: IEEE Computer Society Press, 1998.

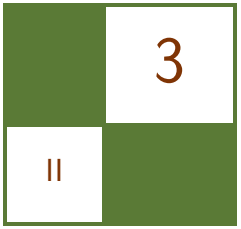
[Pangerl 09] D. Pangerl. "Deferred Rendering Transparency," In *ShaderX*^{7}, pp. 217–224. Hingham, MA: Charles River Media, 2009.

[Patro 07] R. Patro, "Real-Time Approximate Subsurface Scattering," Available at http://www.cs.umd.edu/\simrob/Documents/sss.pdf, 2007.

[Scheuermann 04] T. Scheuermann, "Hair Rendering and Shading." Game Developer's Conference, 2004.

[Wyatt 07] R. Wyatt, "Custom Shaders and Effects." Available at http://www.insomniacgames.com/research_dev/, 2007.

[Valient 07] M. Valient. "Deferred Rendering in Killzone 2." Develop Conference, July 2007.



Large-Scale Terrain Rendering for Outdoor Games

Ferenc Pintér

3.1 Introduction

Visualizing large scale (above 10 km^2) landscapes on current generation consoles is a challenging task, because of the restricted amount of memory and processing power compared with high-end PCs. This article describes in detail how we approach both limitations and deal with the different content creation, rendering, and performance issues. After a short explanation of the decisions and trade-offs we made, terrain-content generation and rendering methods will follow. We conclude by listing the pros and cons of our technique, measured implementation performance, as well as possible extensions.

Figure 3.1. In-game screenshot of a 10 km^2 canyon area. (© 2010 Digital Reality.)

A variety of industry solutions exist for both representing terrain geometry and texturing its surface. Our choices were made with regard to our requirements, which were: a small memory footprint (<20 MB), good rendering performance (<6 ms), easy in-editor (re)painting without UV distortions, and large view distances (>10km), as in Figure 3.1.

3.1.1 Geometry Choice

We opted for mesh-based terrain, which allows for steep, distortion-free slopes and vastly different resolution levels, with complete artist control (as compared with heightfields with vertex-texture fetching, or *render to vertex buffer* (R2VB) [Andersson 07]-heightfields, which, however, can provide in-editor/runtime modifiable geometry). We also chose to store the compressed vertex and triangle data instead of performing on-the-fly mesh construction and the caching, which is sometimes found in planetary rendering engines [Brebion 08, Kemen 08]. Our scale is smaller, and once again we opted for greater artist flexibility.

3.1.2 Texturing Choice

Our solution is based on per-pixel splatting from tiling atlas texture elements, thus it reuses texels over the entire surface of the terrain. This is similar to techniques implemented in other games (*Battlestations: Pacific* [Eidos 08], Figure 3.2, and *Infinity* [Brebion 08]), but instead of using just height- and slope-based rules with additional noise to determine the terrain type at any given pixel, it also relies on precomputed data. This way our artists can paint over the entire terrain, even on uniquely modeled mesh objects. Since the terrain's UVs are unique and relaxed, no distortion appears, even on vertical or slightly overhanging walls.

This method has two main advantages over streaming ultrahigh resolution maps [van Waveren 09, Mittring 08, Barrett 08, van Rossen 08]. First, the required storage space is very low (<15 MB). Second, it does not saturate streaming or *bus transfer bandwidth*. Instant switching between cameras located far from each other is also solved due to the runtime evaluation of shading and texturing. Another advantage is complete artist control of the texturing, which might be more difficult when relying only on procedural or fractal-based methods [Brebion 08, Kemen 08, Eidos 08]. On the other hand, not using unique texture data does result in less variance, though we did not find this to be noticeable.

Extending our asset creation and rendering by using procedural techniques proved to be invaluable. The techniques helped create the basis for various outdoor art assets (foliage, detail object, and terrain texturing) through subtle parameter changes, thus saving time. They also cut memory and bandwidth usage too, emphasizing the fact that GPUs are much faster at doing math than fetching from memory.

Figure 3.2. Screenshot from *Battlestations: Pacific*, released on XBOX 360 and PC. (© 2008 Square Enix Europe.)

3.2 Content Creation and Editing

3.2.1 Workflow

Our terrain assets originate from multiple DCC tools. Artists create the base terrain layout and simple mesh objects with which designers can test level functionality. After this first phase is complete, the base terrain model gets greater morphological and soil-type detail using erosion tools. Artists can iteratively retouch the detailed models if needed, and can also bake *ambient occlusion* (AO) values to the mesh vertices. Parallel to advancing in geometry, textures representing different soil-types get authored and used by the terrain shader. The following steps happen in our in-game editor, after the meshes have been imported from COLLADA format. During this import step, the base terrain gets split into smaller chunks, and *level-of-detail* (LOD) levels are generated. We also use smaller, paintable, and reuseable objects for rock formations, referred to as mesh objects later on. The next step in content creation is additional manual painting for both the base terrain and the mesh objects in the game editor. The finest detail in soil-type information and color tinting is determined in this phase. Finally, mesh and texture assets go through different compression paths for each platform. The complete process is illustrated in Figure 3.3.

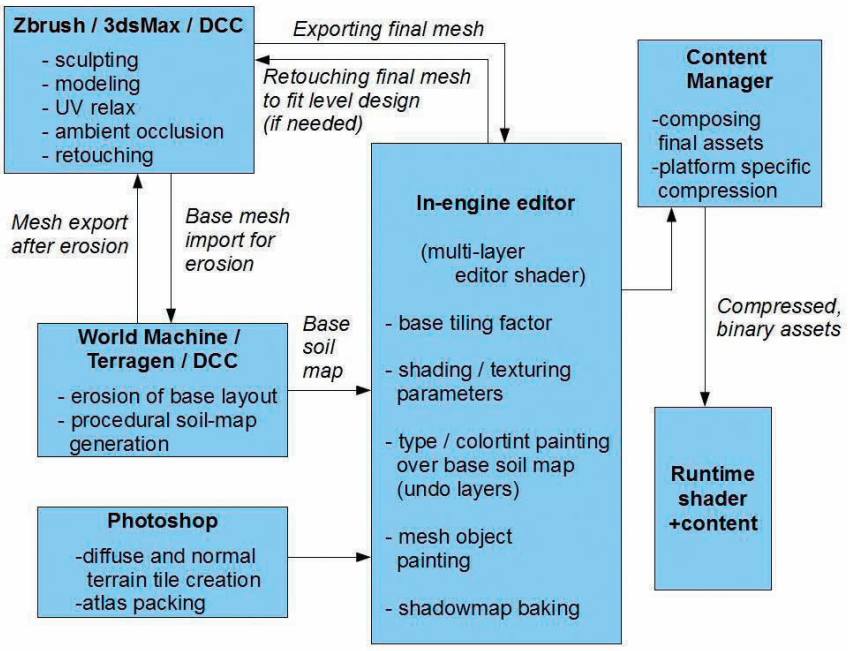


Figure 3.3. Terrain content pipeline/art workflow. (© 2010 Digital Reality.)

3.2.2 Determining Soil Type

Multiple approaches may be used to decide which soil type to apply to a given region of the terrain. Since this information does not change during gameplay it may be precomputed and stored offline.

3.2.3 Procedural Rules Stored in a Lookup Table

A *lookup table* (LUT) may be used to determine which terrain type shall occur at different terrain slope and height values. The LUT can be stored as a two-dimensional texture parameterized by terrain slope and height along the u and v axes. Addressing the table at runtime requires using slope-height pairs, interpolated data coming from the vertex shader [Eidos 08, Brebion 08]. Its advantage is fast iteration times, and simple implementation, though it has its drawbacks too. Because the LUT is applied globally to the terrain, it does not allow the artists to have local control over the terrain's texturing. Moreover, because the LUT is completely decoupled from the position of terrain in the game world, we cannot store local shading or color-tint information such as local soil types in it.

We found these drawbacks too restricting, and chose to go with the approaches listed below.

3.2.4 Procedural and Manual Painting Stored in a UV-Space Tilemap

Alternatively, we can use tile-index texture that covers the entire terrain and can be sampled using the unique, relaxed UV. In our case, we used a 4-channel map, encoding a color-tint value in three channels (to break repetitive patterns), and the terrain-type index in the fourth (see Figure 3.4). This method has multiple advantages over the first: it can be locally controlled by art needs, separate regions within the map can be edited concurrently by multiple artists, and it can also use procedural methods as its basis. The only drawback is that is uses a fixed resolution for terrain-type data, but this never proved to be a problem for us.

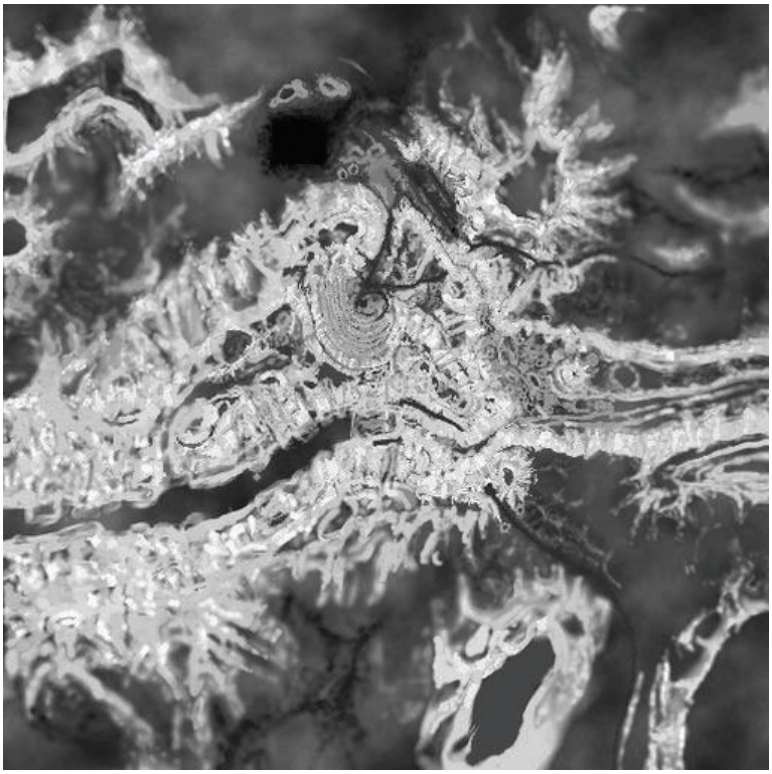


Figure 3.4. Alpha channel of the painted 512^2 tilemap, containing terrain-type info. (© 2010 Digital Reality.)

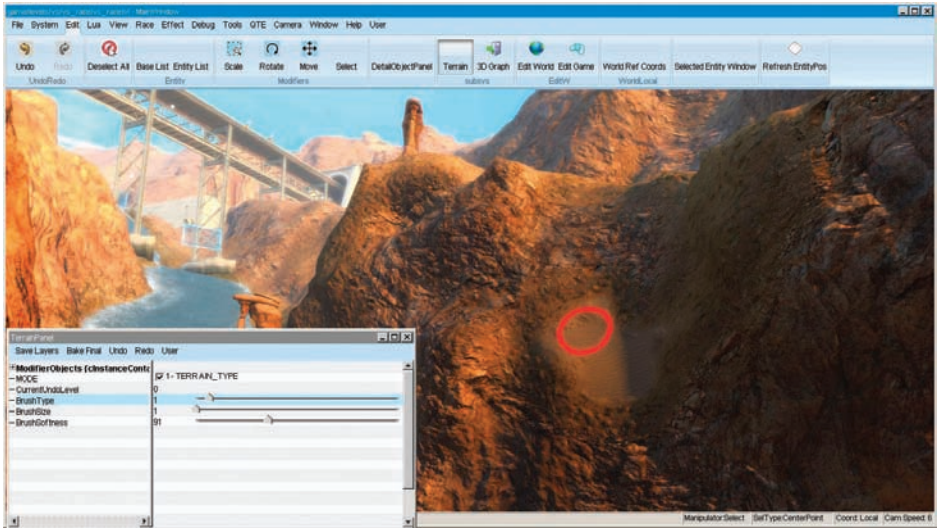


Figure 3.5. Editor-artist interface for painting the tilemap, red ring indicating brush position. (© 2010 Digital Reality.)

Creating the tilemap is very intuitive. Its terrain-index part can be based on either soil-type maps exported from many commercial terrain-generation/erosion software programs (though you might need to convert the world-space type values to relaxed UV space), or global terrain height- and slope-based rules, enhanced by noise.

The base for the color part can again originate from DCC tools, from shadow maps, or can be simply desaturated color noise. Over this base, artists can easily paint or modify the chosen terrain-type values using simple brushes, and temporary layers. Ray casting is used to determine which tilemap texels the brush touches. The editor interface (see Figure 3.5) can also support multiple undo-levels (by caching paint commands), soft brushes, or paint limitations (to allow painting only over regions within desired height/slope limits).

At runtime, hardware bilinear filtering of the tilemap indices automatically solves type-blending problems present in the LUT method, and different nearby tile-index values will get smoothly interpolated over the terrain pixels. We can also compress the tilemap using DXT5 texture compression. Since this format compresses the three color channels independently from the alpha channel, we can keep most of the index resolution while shrinking memory usage.

Note: Suppose we have sand encoded as tile-index value 0, grass as 1, and rock as 2. Now, due to filtering, rock can never be visible near sand, but only through an intermediate grass area. This can be avoided by duplicating types with

different neighbors in the atlas, backed by a bit more complex atlas-sampling math. We did not need this, however.

3.2.5 Procedural and Manual Painting Stored in Mesh Vertices

We can also detach geometry containing holes, overhangs, twists, or other hard-to-relax regions from the general terrain mesh, or create any geometry and use it as a paintable terrain-mesh object. Apart from better fitting into the base terrain, we can spare quite a lot of texture memory by using shared atlases and a slightly modified terrain shader here too, instead of unique maps. Artists can UV-map the meshes using arbitrary methods and DCC tools, producing a nonoverlapping unique UV, with seams and connections moved to less noticeable areas.

DCC tools do not have direct view of relative asset extents and spatial proportions inside the game. To help artists with UV mapping and proper tiling factors, we multiply the base UV during editor import with a constant value. This value is determined by the total geometrical surface of the object divided by its total UV-space surface. This way, tiling matches perfectly on all objects and the terrain, no matter how much UV space is used inside the $0 \ldots 1$ region for the given object in the DCC tool. Another feature implemented to help artists is that which allows already-painted mesh objects to reuse their respective paintings if their geometry gets modified again in the DCC tools. This functionality stores the painting of every object also as a texture that gets saved from the editor automatically, along with mesh data. It is used only in the editor, and its texels contain UV and soil-type information, one texel for every vertex. Reapplying paintings is merely finding a UV match between the vertices of the modified mesh, and the saved texels. If the majority of vertices kept their original UVs after the DCC mesh modification, most painting information can be reused.

After geometry authoring is done for a mesh, its procedural and manual painting follows in the editor. Manual painting works similarly to painting the base terrain (illustrated in Figure 3.6). By using ray casting, we can figure out which object, thus which shared-mesh vertex buffer to modify, and with a simple vertex-distance-based spherical three-dimensional brush, artists can encode soil-type information into the mesh vertices. (This can sometimes be hidden for free in an unused byte of a compressed vertex, like in the w component of position or normal data.)

Soil-type continuity where the mesh objects meet the terrain is almost as important as matching normals near the connection region. Because mesh paintings are shared between instances of the same object, and terrain painting is unique due to the tilemap, the latter can be easily retrofitted to match the objects at the connection region. Also, by using a second pair of diffuse/normal atlases for the mesh objects, (containing only a few redundant tiles from the main terrain atlas for connections) greater soil variance can be achieved. Because of complete UV control, we can use tiles that have dominant directional features too.

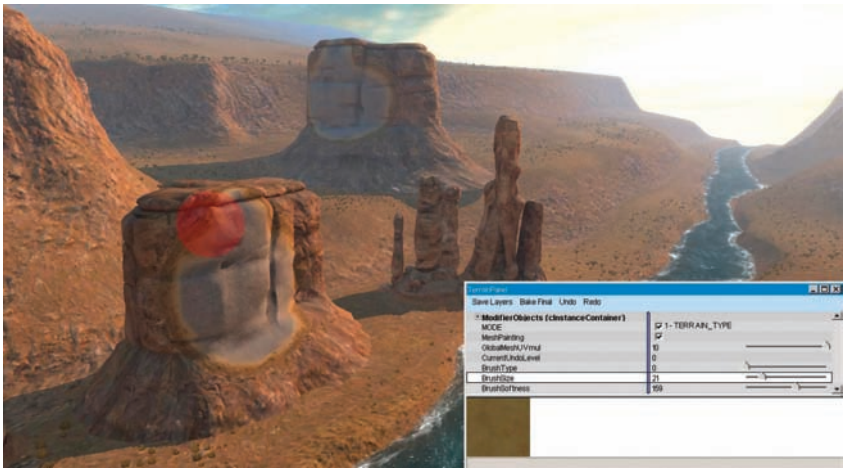


Figure 3.6. Editor-artist interface for painting mesh objects, red sphere indicating brush volume. (© 2010 Digital Reality.)

On the base terrain, UV relaxing causes tiles to be mapped in different directions based on which side of a hill they are on—UV derivatives change directions with respect to world-space coordinates—thus tiles with heavy direction dependency might be used properly on only one side of a hill, or by putting multiple rotated versions in the atlas.

If required, color-tint or luminance information such as AO can also be painted (or computed in-editor) and baked into mesh vertices.

3.3 Runtime Shading

The base of the runtime rendering solution is texture splatting on a per-pixel level. Using some per-pixel input data, and a unique, relaxed UV channel, the shader can sample different tiling-terrain textures, and blend among them.

To reduce the count of textures the shader needs to sample from, these tiling textures—corresponding to different soil types—can be packed into textures atlases (see Figure 3.7), or texture arrays on XBOX 360 and DX10/11 architectures [Brebion 08, Wloka 03]. Care shall be taken when generating miplevels for the atlas though, as the individual tile-mipmap texels must not get mixed with their neighbors. Creating the compressed and mipmapped tiles first, and then packing them to an atlas is one solution. Anisotropic filtering also becomes more complex when using atlases [van Waveren 09].

If we choose the atlas method, we want the tiling to wrap around in a smaller UV region (hence this is sometimes referred to as subtiling), say $0 \ldots 0.25$ if

Figure 3.7. A packed 2048^2 diffuse-texture atlas, containing 16 different terrain types. (© 2010 Digital Reality.)

we have a 4×4 atlas. This also means that we cannot rely on hardware texture wrapping; this must be performed manually in the shader. As we will see, this causes problems in hardware miplevel selection (texture arrays do not need these corrections, however). For this to work correctly, one must know how the hardware calculates which mip levels to use. GPUs use the first derivatives of screen-space texture coordinates in any given 2×2 pixel block and the dimensions of the texture itself to determine the pixel-to-texel projection ratio, and thus find the appropriate miplevel to use. To access a tile from the atlas for any pixel, we need to emulate the hardware wrapping for the tile. By using the `frac() hlsl`

intrinsic, we break the screenspace UV derivative continuity for the pixel quads at tile borders. Since the derivatives will be very large, the hardware will pick the largest miplevel from the chain, which in turn results in a one-pixel-wide seam whenever the subtiling wraps around. Fortunately, we have many options here: we can balance the GPU load between texture filtering, *arithmetic logic unit* (ALU) cost, shader thread counts, texture stalls, and overall artifact visibility.

The safest but slowest option is to calculate the mip level manually in the shader, right before sampling from the atlas [Wloka 03,Brebion 08]. This produces the correct result, but the extra ALU cost is high since we need to issue gradient instructions that require extra GPU cycles, and textures need to be sampled with manually specified mip levels, which reduces the sampling rate on many architectures. As a side effect, texture stalls begin to appear in the pipeline. We can use multiple methods to shorten these stalls. Some compilers and platforms allow for explicitly setting the maximum number of general purpose *GPU registers* (GPRs) a compiled shader can use. (They try to optimize the shader code to meet the specified limit, sometimes by issuing more ALUs to move temporary shader data around with fewer registers.) If a shader uses fewer GPRs, more shading cores can run it in parallel, thus the number of simultaneous threads increases. Using more threads means that stalling all of them is less probable. On systems using unified shader architectures, one can also increase pixel shader GPR count by reducing the available GPRs for the vertex shader. Some platforms also have shader instructions that explicitly return the mip level the hardware would use at a given pixel, thus saving you from having to compute it yourself in a shader. Using dynamic branching and regular hardware mipmapping on pixel quads far from the `frac() region` as speedup might also prove useful.

Branch performance might degrade for faraway fragments though, where tiling UV values and derivatives vary fast, and pixels in the same GPU-pixel-processing vector must take different branches. Branching might be disabled for distant fragments, since stall problems are also most relevant on up close terrain, which fills most screen area and uses the first few mip levels.

One option for estimating the correct mip level is to construct a texture that encodes the mip index in the texture itself (for example, the first mip level encodes "0" in all its texels, the second mip level encodes "1" in all its texels, etc.). This texture should have the same dimensions as the atlas tile. You can then use a normal texture fetch to sample this texture and allow the hardware to choose the appropriate mip level. The value of the texel that is fetched will indicate which mip level was chosen by the hardware and then can be used to issue a `tex2dlod` instruction on the atlas tile. Dynamic branching is a viable option here too.

We chose to go with a third option, which is the fastest, but does result in some minor artifacts which we deemed acceptable. We simply sample using regular `tex2D`, but we generate only the first four mipmaps of the mip chain. This means that the GPU has to filter a bit more, but our measurements have shown that only 7–10% of texels fall into the larger miplevels, thus the performance

Figure 3.8. Screenshot from the canyon, with 100x UV tiling factor (note the lack of UV distortions on walls). (© 2010 Digital Reality.)

impact is considered to be minimal compared to using reduced-speed `tex2Dlods`. The visual artifacts are minimized because the seam will always use the fourth mip level, and colors differ much less between the first and last levels. We also switched to texture atlases of 1×16 dimensions instead of 4×4, thus we can use hardware texture wrapping for one direction, halving mip level errors arising from using `frac()`, while also using fewer ALU operations to address the atlases.

At this point, we have diffuse and normal atlases ready to be sampled in the shader. To improve quality, we blend between two nearby tiles—using the fractional part of the interpolated tile index—by reading twice from the same atlas, with respective UV offsets. Shading the terrain and the painted meshes is identical, and is based on simple per-pixel lambertian diffuse, and hemispherical ambient terms. Shadow contribution is composed of a lookup into a precomputed and blurred static-shadow map, cross-fading with a cascaded dynamic and blurred *exponential shadow map* (ESM), and AO baked into vertices. There are many ways to experiment with more complex lighting models, however, correctly set atmospheric settings, fog, *high dynamic range* (HDR), fake scattering [Quilez 09], and soil-type maps can provide a solid impression already. See Figure 3.8 for reference.

For your convenience, the runtime per-pixel, texture-splatting shader code is listed in Listing 3.1. Note that editor-mode paint layers, brush-ring-overlay paint functionality, vertex unpacking, shadows, and fog calculations are omitted for clarity.

```
//Runtime terrain shader with tilemap-based per-pixel
//splatting using atlases (tangent-space lighting).

static const float TILEMAP_SIZE= 512.0f;
static const float TILES_IN_ROW= 4.0f;
static const float MAX_TILE_VALUE= TILES_IN_ROW*TILES_IN_ROW-1;

struct sVSInput
{
    float4 Position : POSITION;
    float3 Normal   : NORMAL;
    float2 UV       : TEXCOORD0;
    float3 Tangent  : TEXCOORD1;
};

struct sVSOutput
{
    float4 ProjPos     : POSITION;
    float2 UV          : TEXCOORD0;
    float3 Normal      : TEXCOORD1;
    float3 TgLightVec  : TEXCOORD2;
};

float4x3 cWorldMatrix;
float4x4 cViewProjMatrix;
float cUVmultiplier; //Terrain-texture tiling factor.
float3 cCameraPos;
float3 cSunDirection;
float3 cSunColor;

//Lighting is in tangent space.
float3x3 MakeWorldToTangent(float3 iTangent, float3 iNormal)
{
    float3x3 TangentToLocal=
        float3x3(iTangent, cross(iNormal,iTangent), iNormal);
    float3x3 TangentToWorld=
        mul(TangentToLocal,(float3x3)cWorldMatrix);
    float3x3 WorldToTangent = transpose(TangentToWorld);

    return WorldToTangent;
}

sVSOutput vpmain(sVSInput In)
{
    sVSOutput Out;

    float3 WorldPos= mul(In.Position, cWorldMatrix);
    Out.ProjPos= mul(float4(WorldPos,1), cViewProjMatrix);

    Out.Normal= mul(In.Normal.xyz, (float3x3)cWorldMatrix);

    Out.UV= In.UV * cUVmultiplier;
```

```
    float3x3 WorldToTangent=
        MakeWorldToTangent(In.Tangent, In.Normal);

    Out.TgLightVec= mul(cSunDirection.xyz, WorldToTangent);

    return Out;
}

sampler2D DiffAtlas;
sampler2D NormAtlas;
sampler2D TileTable;

float GetMipLevel(float2 iUV, float2 iTextureSize)
{
    float2 dx= ddx(iUV * iTextureSize.x);
    float2 dy= ddy(iUV * iTextureSize.y);
    float d= max( dot(dx, dx), dot(dy, dy) );
    return 0.5 * log2(d);
}

float4 fpmain(sVSOutput In) : COLOR
{
    float4 TileMapTex= tex2D(TileTable, In.UV/cUVmultiplier);
    float3 ColorTint= TileMapTex.xyz;
    float TileIndex= TileMapTex.w * MAX_TILE_VALUE;

    float MIP= GetMipLevel(In.UV, TILEMAP_SIZE.xx);

    float2 fracUV = frac(In.UV);
    float2 DiffCorrectUV= fracUV/4.0f;

    //Blend types and blend ratio.
    float type_A = floor(TileIndex);
    float type_B = ceil(TileIndex);
    float factor = TileIndex - type_A;

    float tmp = floor(type_A/4);
    float2 UV_A = DiffCorrectUV + float2(type_A-tmp*4,tmp)/4;
    tmp = floor(type_B/4);
    float2 UV_B = DiffCorrectUV + float2(type_B-tmp*4,tmp)/4;

    // 2 Lookups needed, for blending between layers.
    float4 colA= tex2Dlod(DiffAtlas, float4(UV_A,0,MIP));
    float4 colB= tex2Dlod(DiffAtlas, float4(UV_B,0,MIP));

    float4 DiffuseColor= lerp(colA, colB, factor);

    float4 normA= tex2Dlod(NormAtlas, float4(UV_A,0,MIP));
    float4 normB= tex2Dlod(NormAtlas, float4(UV_B,0,MIP));

    float4 normtex= lerp(normA, normB, factor);

    //Extract normal map.
    float3 norm= 2*(normtex.rgb-0.5);
```

```
    float3 tgnormal= normalize(norm);

    float NdotL=
        saturate(dot(tgnormal, normalize(In.TgLightVec)));
    float3 SunDiffuseColor= NdotL * cSunColor;
    float3 Albedo= DiffuseColor.xyz * ColorTint * 2;
    float3 AmbientColor= 0.5;

    float3 LitAlbedo= Albedo * (AmbientColor + SunDiffuseColor);
    return float4(LitAlbedo,1);
}
```

Listing 3.1. Simplified runtime shaders including manual mipmap computation.

3.4 Performance

The system load for rendering the base terrain mesh spanning a 10 km^2 area, and consisting of 600 k triangles is 14 MB memory, and 6 ms frame time, on XBOX 360. This includes geometry vertex and index data, LODs, diffuse, normal, shadow, and terrain type maps, while not using any streaming bandwidth. To reach the desired rendering time and memory footprint, a couple of optimizations are required.

To allow using 16-bit indices in the triangle list, the terrain had to be sliced into blocks no larger than 65 k vertices, during COLLADA import. Using blocks with <1 km^2 area also helps our static KD-tree based culling. Balancing between better culling and fewer draw calls can be done by adjusting block count and block size. LODs are also generated at import time, since they are essential to reduce vertex load, and also keep the pixel quad efficiency high. If the LODs are generated by skipping vertices, only index data needs to be extended by a small amount (say, 33%, if each consecutive level contains a quarter of the previous version, with regard to triangle count), and each new LOD can refer to the original, untouched vertex buffer. Keeping silhouettes and vertices at block boundaries regardless of LOD are important in order to prevent holes from appearing in places where differed LODs meet. To help pre- and post-transform vertex and index caches, vertex and index reordering is also done at the import phase.

Vertex compression is also heavily used to reduce the required amount of memory and transfer bandwidth. We can store additional information in the vertices. AO and soil-type indices are valid choices, but color-tint values, shadow terms, or bent normals for better shading are still possible.

Texture atlases are also compressed. We found atlases of 4×4 512^2 tiles to contain enough variation and resolution too. The diffuse atlas can use the DXT1 format, while the normal atlas can use better, platform-specific 2-channel

compressed formats where available. The tilemap can also be compressed to DXT5, keeping most information of the tile indices in the alpha channel, while heavily compressing color-tint data. The static shadow map can also be compressed to DXT1, or luminance of multiple shadow texels can be packed into one texel of a two-channel map. (ATI1N (BC4/DXT5A), ATI2N (BC5/3Dc), DXN, and CTX1 formats are easy choices.) Using floating-point depth buffers and depth pre-pass for rejecting pixels, and occlusion queries or conditional rendering to entirely skip draw calls can also prove to be helpful.

3.5 Possible Extensions

Many aspects of the described system can be improved or extended. Some of the methods listed below are general improvements, some are platform specific, while some trade flexibility for faster rendering or less resource usage.

Heightmap-based geometry can provide a more compact, though computationally more expensive representation of the terrain. Representing steep slopes without texture distortions might be possible if we store three component-position offsets of vertices of relaxed UV space grid, like mesh textures. More aggressive vertex compression is also still possible, even without using heightmaps. Using more blocks of terrain with smaller block sizes, or using triplanar mapping [Geiss 07] might yield better compression opportunities.

Texturing can be enhanced in multiple ways, though most of them put more texture fetch and filtering burden on the GPU. Using a large tiling factor can either result in visible tiling of soil types, or not having the ability to visualize larger terrain features, originating from the atlas. A solution is to sample the atlases twice, with two very different tiling factors, and cross fade their color and normal data. This enhances large details in the far regions, lending unique character to the terrain, while it provides detail up close where tiling is less visible. Large details dissolve as the camera moves close, but this is barely noticeable. Figure 3.9 illustrates the concept.

Another option is reducing the tiling factor, and blending in desaturated detail textures up close. This allows a choice among different detail maps, based on the general terrain soil-type. Using noise (defined by interpolated height and slope data) to perturb the soil-type on per-pixel level in the shader with ALU instructions is also a possibility to reduce tiling or add up close detail. The inverse of this method is also viable, and has been the de facto standard for terrain texturing for many years. We can stream a large, unique, diffuse, and normal map for far details, and blend in the per-pixel splatting as close-up or mid-range detail. Transition regions between different soil types can also be enhanced by using blend maps, which can be efficiently packed into the alpha channel of the diffuse atlas [Hardy 09]]. Yet another option is to use more soil-type textures in the atlas to give more variance. Soil-type information can also

Figure 3.9. Screenshot of a canyon wall, with distance-dependant atlas tiling factor. (© 2010 Digital Reality.)

drive collision-particle generation, or procedural placement of detail object and foliage [Andersson 07].

We can use ambient aperture lighting to give bump maps some shadowing [Persson 06], and image-based ambient and diffuse lighting using bent normals. Specular reflections can also be added where surfaces are wet, for example, riverbanks [Geiss 07]. Atmospheric scattering with volumetric light/fog (or faking it [Quilez 09]) is also an option to enhance realism of the rendered terrain. Enhancing normal mapping by using more complex parallax effects might also be feasible.

Finally, caching previously computed fragment data (composed albedo or final normal data), either in screen space, or around the camera in relaxed UV space can speed up rendering [Herzog et al. 10].

3.6 Acknowledgments

First, I would like to thank Balázs Török, my friend and colleague, for his helpful suggestions and for the initial idea of contributing to the book. I would also like to thank the art team at Digital Reality for their excellent work, on which I based my illustrations. Also, special thanks to my love Tímea Kapui for her endless patience and support.

Bibliography

[Andersson 07] Johan Andersson. "Terrain Rendering in Frostbite Using Procedural Shader Splatting." In *ACM SIGGRAPH 2007 courses*. New York: ACM, 2007. Available online (http://ati.amd.com/developer/SIGGRAPH07/Chapter5-Andersson-Terrain_Rendering_in_Frostbite.pdf).

[Barrett 08] Sean Barrett. "Sparse Virtual Textures blog." 2008. Available online (http://www.silverspaceship.com/src/svt/).

[Brebion 08] Flavien Brebion. "Journal of Ysaneya, gamedev.net." 2008. Available online (http://www.gamedev.net/community/forums/mod/journal/journal.asp?jn=263350).

[Dudash 07] Bryan Dudash. "DX10 Texture Arrays, nVidia SDK." 2007. Available online (http://developer.download.nvidia.com/SDK/10/Samples/TextureArrayTerrain.zip).

[Eidos 08] Eidos, Square Enix Europe. "Battlestations: Pacific." 2008. Available online (http://www.battlestations.net).

[Geiss 07] Ryan Geiss. "Generating Complex Procedural Terrains Using the GPU." In *GPU Gems 3*. Reading, MA: Addison Wesley, 2007. Available online (http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html).

[Hardy 09] Alexandre Hardy. "Blend Maps: Enhanced Terrain Texturing." Available online (http://www.ahardy.za.net/Home/blendmaps).

[Herzog et al. 10] Robert Herzog, Elmar Eisemann, Karol Myszkowski, and H-P. Seidel. "Spatio-Temporal Upsampling on the GPU."

[Kemen 08] Brano Kemen. "Journal of Lethargic Programming, Cameni's gamedev.net blog, July 2008." 2008. Available online (http://www.gamedev.net/community/forums/mod/journal/journal.asp?jn=503094&cmonth=7&cyear=2008).

[Mittring 08] Martin Mittring. "Advanced Virtual Texturing Topics." In *ACM SIGGRAPH 2008 Classes SIGGRAPH '08*, pp. 23–51. New York, NY, USA: ACM, 2008.

[Pangilinan and Ruppel 10] Erick Pangilinan and Robh Ruppel. "Uncharted 2 Art direction." *GDC 2010*. Available online (http://www.gdcvault.com/free/category/280/conference/).

[Persson 06] Emil Persson. "Humus, Ambient Aperture Lighting." Available online (http://www.humus.name/index.php?page=3D&ID=71).

[Quilez 09] Inigo Quilez. "Behind Elevated." *Function 2009*. Available online (http://www.iquilezles.org/www/material/function2009/function2009.pdf).

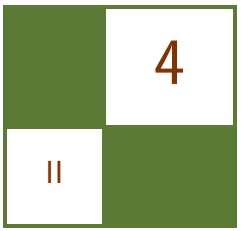
[Schueler 06] Christian Schueler. "Normal Mapping without Pre-Computed Tangents." In *ShaderX*^{5}*: Advanced Rendering Techniques*, Chapter II.6. Hingham, MA: Charles River Media, 2006. Available online (http://www.shaderx5.com/TOC.html).

[van Rossen 08] Sander van Rossen. "Sander's Blog on Virtual Texturing." 2008. Available online (http://sandervanrossen.blogspot.com/2009/08/virtual-texturing-part-1.html).

[van Waveren 09] J. M. P. van Waveren. "idTech5 Challeges: From Texture Virtualization to Massive Parallelization." *Siggraph 2009*. Available online (http://s09.idav.ucdavis.edu/talks/05-JP_id_Tech_5_Challenges.pdf).

[Wetzel 07] Mikey Wetzel. "Under The Hood: Revving Up Shader Performance." *Microsoft Gamefest Unplugged Europe, 2007*. Available online (http://www.microsoft.com/downloads/details.aspx?FamilyId=74DB343E-E8FF-44E9-A43E-6F1615D9FCE0&displaylang=en).

[Wloka 03] Matthias Wloka. "Improved Batching Via Texture Atlases." In *ShaderX*^{3}. Hingham, MA: Charles River Media, 2003. Available online (http://www.shaderx3.com/Tables%20of%20Content.htm).



Practical Morphological Antialiasing

Jorge Jimenez, Belen Masia, Jose I. Echevarria, Fernando Navarro, and Diego Gutierrez

The use of antialiasing techniques is crucial when producing high quality graphics. Up to now, *multisampling antialiasing* (MSAA) has remained the most advanced solution, offering superior results in real time. However, there are important drawbacks to the use of MSAA in certain scenarios. First, the increase in processing time it consumes is not negligible at all. Further, limitations of MSAA include the impossibility, in a wide range of platforms, of activating multisampling when using *multiple render targets* (MRT), on which fundamental techniques such as deferred shading [Shishkovtsov 05, Koonce 07] rely. Even on platforms where MRT and MSAA can be simultaneously activated (i.e., DirectX 10), implementation of MSAA is neither trivial nor cost free [Thibieroz 09]. Additionally, MSAA poses a problem for the current generation of consoles. In the case of the Xbox 360, memory constraints force the use of CPU-based tiling techniques in case high-resolution frame buffers need to be used in conjunction with MSAA; whereas on the PS3 multisampling is usually not even applied. Another drawback of MSAA is its inability to smooth nongeometric edges, such as those resulting from the use of alpha testing, frequent when rendering vegetation. As a result, when using MSAA, vegetation can be antialiased only if alpha to coverage is used. Finally, multisampling requires extra memory, which is always a valuable resource, especially on consoles.

In response to the limitations described above, a series of techniques have implemented antialiasing solutions in shader units, the vast majority of them being based on edge detection and blurring. In *S.T.A.L.K.E.R.* [Shishkovtsov 05], edge detection is performed by calculating differences in the eight-neighborhood depth values and the four-neighborhood normal angles; then, edges are blurred using a cross-shaped sampling pattern. A similar, improved scheme is used in *Tabula Rasa* [Koonce 07], where edge detection uses threshold values that are resolution

independent, and the full eight-neighborhood of the pixel is considered for differences in the normal angles. In *Crysis* [Sousa 07], edges are detected by using depth values, and rotated triangle samples are used to perform texture lookups using bilinear filtering. These solutions alleviate the aliasing problem but do not mitigate it completely. Finally, in *Killzone 2*, samples are rendered into a double horizontal resolution G-buffer. Then, in the lighting pass, two samples of the G-buffer are queried for each pixel of the final buffer. The resulting samples are then averaged and stored in the final buffer. However, this necessitates executing the lighting shader twice per final pixel.

In this article we present an alternative technique that avoids most of the problems described above. The quality of our results lies between $4x$ and $8x$ MSAA at a fraction of the time and memory consumption. It is based on morphological antialiasing [Reshetov 09], which relies on detecting certain image patterns to reduce aliasing. However, the original implementation is designed to be run in a CPU and requires the use of list structures that are not GPU-amenable.

Since our goal is to achieve real-time practicality in games with current mainstream hardware, our algorithm implements aggressive optimizations that provide an optimal trade-off between quality and execution times. Reshetov searches for specific patterns (U-shaped, Z-shaped, and L-shaped patterns), which are then decomposed into simpler ones, an approach that would be impractical on a GPU. We realize that the pattern type, and thus the antialiasing to be performed, depends on only four values, which can be obtained for each edge pixel (*edgel*) with only two memory accesses. This way, the original algorithm is transformed so that it uses texture structures instead of lists (see Figure 4.1). Furthermore, this approach allows handling of all pattern types in a symmetric way, thus avoiding the need to decompose them into simpler ones. In addition, precomputation of

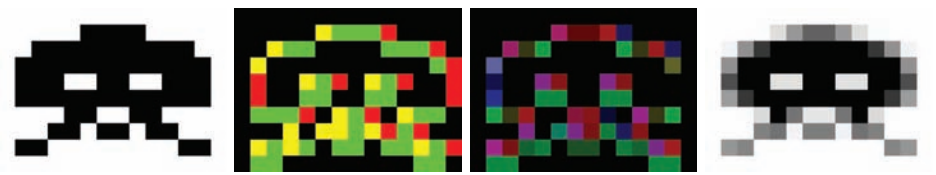


Figure 4.1. Starting from an aliased image (left), edges are detected and stored in the edges texture (center left). The color of each pixel depicts where edges are: green pixels have an edge at their top boundary, red pixels at their left boundary, and yellow pixels have edges at both boundaries. The edges texture is then used in conjunction with the precomputed area texture to produce the blending weights texture (center right) in the second pass. This texture stores the weights for the pixels at each side of an edgel in the RGBA channels. In the third pass, blending is performed to obtain the final antialiased image (right).

certain values into textures allows for an even faster implementation. Finally, in order to accelerate calculations, we make extensive use of hardware bilinear interpolation for smartly fetching multiple values in a single query and provide means of decoding the fetched values into the original unfiltered values. As a result, our algorithm can be efficiently executed by a GPU, has a moderate memory footprint, and can be integrated as part of the standard rendering pipeline of any game architecture.

Some of the optimizations presented in this work may seem to add complexity at a conceptual level, but as our results show, their overall contribution makes them worth including. Our technique yields image quality between $4x$ and $8x$ MSAA, with a typical execution time of 3.79 ms on Xbox 360 and 0.44 ms on a NVIDIA GeForce 9800 GTX+, for a resolution of 720p. Memory footprint is $2x$ the size of the backbuffer on Xbox 360 and $1.5x$ on the 9800 GTX+. According to our measurements, $8x$ MSAA takes an average of 5 ms per image on the same GPU at the same resolution, that is, our algorithm is $11.80x$ faster.

In order to show the versatility of our algorithm, we have implemented the shader both for Xbox 360 and PC, using DirectX 9 and 10 respectively. The code presented in this article is that of the DirectX 10 version.

4.1 Overview

The algorithm searches for patterns in edges which then allow us to reconstruct the antialiased lines. This can, in general terms, be seen as a revectorization of edges. In the following we give a brief overview of our algorithm.

First, edge detection is performed using depth values (alternatively, luminances can be used to detect edges; this will be further discussed in Section 4.2.1). We then compute, for each pixel belonging to an edge, the distances in pixels from it to both ends of the edge to which the edgel belongs. These distances define the position of the pixel with respect to the line. Depending on the location of the edgel within the line, it will or will not be affected by the antialiasing process. In those edges which have to be modified (those which contain yellow or green areas in Figure 4.2 (left)) a blending operation is performed according to Equation (4.1):

$$c_{\text{new}} = (1 - a) \cdot c_{\text{old}} + a \cdot c_{\text{opp}}, \tag{4.1}$$

where c_{old} is the original color of the pixel, c_{opp} is the color of the pixel on the other side of the line, c_{new} is the new color of the pixel, and a is the area shown in yellow in Figure 4.2 (left). The value of a is a function of both the pattern type of the line and the distances to both ends of the line. The pattern type is defined by the *crossing edges* of the line, i.e., edges which are perpendicular to the line and thus define the ends of it (vertical green lines in Figure 4.2). In order to save processing time, we precompute this area and store it as a two-channel texture that can be seen in Figure 4.2 (right) (see Section 4.3.3 for details).

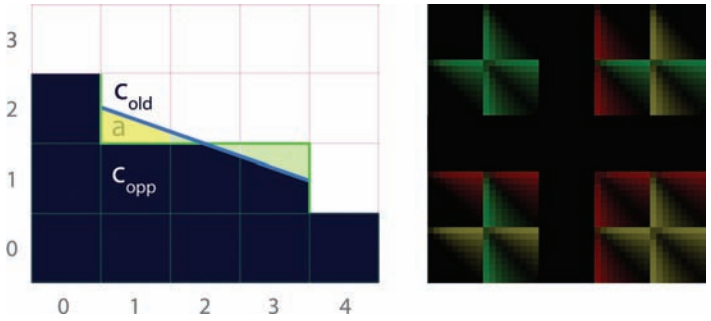


Figure 4.2. Antialiasing process (left). Color c_{opp} bleeds into c_{old} according to the area a below the blue line. Texture containing the precomputed areas (right). The texture uses two channels to store areas at each side of the edge, i.e., for a pixel and its opposite (pixels $(1, 1)$ and $(1, 2)$ on the left). Each 9×9 subtexture corresponds to a pattern type. Inside each of these subtextures, (u, v) coordinates encode distances to the left and to the right, respectively.

The algorithm is implemented in three passes, which are explained in detail in the following sections. In the first pass, edge detection is performed, yielding a texture containing edges (see Figure 4.1 (center left)). In the second pass the corresponding blending weight[1] (that is, value a) for each pixel adjacent to the edge being smoothed is obtained (see Figure 4.1 (center right)). To do this, we first detect the pattern types for each line passing through the north and west boundaries of the pixel and then calculate the distances of each pixel to the *crossing edges*; these are then used to query the precomputed area texture. The third and final pass involves blending each pixel with its four-neighborhood using the blending weights texture obtained in the previous pass.

The last two passes are performed separately to spare calculations, taking advantage of the fact that two adjacent pixels share the same edgel. To do this, in the second pass, pattern detection and the subsequent area calculation are performed on a per-edgel basis. Finally, in the third pass, the two adjacent pixels will fetch the same information.

Additionally, using the *stencil buffer* allows us to perform the second and third passes only for the pixels which contain an edge, considerably reducing processing times.

4.2 Detecting Edges

We perform edge detection using the depth buffer (or luminance values if depth information is not available). For each pixel, the difference in depth with respect to the pixel on top and on the left is obtained. We can efficiently store the edges

[1] Throughout the article *blending weight* and *area* will be used interchangeably.

for all the pixels in the image this way, given the fact that two adjacent pixels have a common boundary. This difference is thresholded to obtain a binary value, which indicates whether an edge exists in a pixel boundary. This threshold, which varies with resolution, can be made resolution independent [Koonce 07]. Then, the left and top edges are stored, respectively, in the red and green channels of the edges texture, which will be used as input for the next pass.

Whenever using depth-based edge detection, a problem may arise in places where two planes at different angles meet: the edge will not be detected because of samples having the same depth. A common solution to this is the addition of information from normals. However, in our case we found that the improvement in quality obtained when using normals was not worth the increase in execution time it implied.

4.2.1 Using Luminance Values for Edge Detection

An alternative to depth-based edge detection is the use of luminance information to detect image discontinuities. Luminance values are derived from the CIE XYZ (color space) standard:

$$L = 0.2126 \cdot R + 0.7152 \cdot G + 0.0722 \cdot B.$$

Then, for each pixel, the difference in luminance with respect to the pixel on top and on the left is obtained, the implementation being equivalent to that of depth-based detection. When thresholding to obtain a binary value, we found 0.1 to be an adequate threshold for most cases. It is important to note that using either luminance- or depth-based edge detection does not affect the following passes.

Although qualitywise both methods offer similar results, depth-based detection is more robust, yielding a more reliable edges texture. And, our technique takes, on average, 10% less time when using depth than when using luminance values. Luminance values are useful when depth information cannot be accessed and thus offer a more universal approach. Further, when depth-based detection is performed, edges in shading will not be detected, whereas luminance-based detection allows for antialias shading and specular highlights. In general terms, one could say that luminance-based detection works in a more perceptual way because it smoothes *visible* edges. As an example, when dense vegetation is present, using luminance values is faster than using depth values (around 12% faster for the particular case shown in Figure 4.5 (bottom row)), since a greater number of edges are detected when using depth values. Optimal results in terms of quality, at the cost of a higher execution time, can be obtained by combining luminance, depth, and normal values.

Listing 4.1 shows the source code of this pass, using depth-based edge detection. Figure 4.1 (center left) is the resulting image of the edge-detection pass, in this particular case, using luminance-based detection, as depth information is not available.

```
float4 EdgeDetectionPS(float4 position: SV_POSITION,
                       float2 texcoord: TEXCOORD0): SV_TARGET {

  float D = depthTex.SampleLevel(PointSampler,
                                 texcoord, 0);
  float Dleft = depthTex.SampleLevel(PointSampler,
                                     texcoord, 0, -int2(1, 0));
  float Dtop  = depthTex.SampleLevel(PointSampler,
                                     texcoord, 0, -int2(0, 1));

  // We need these for updating the stencil buffer.
  float Dright  = depthTex.SampleLevel(PointSampler,
                                       texcoord, 0, int2(1, 0));
  float Dbottom = depthTex.SampleLevel(PointSampler,
                                       texcoord, 0, int2(0, 1));

  float4 delta = abs(D.xxxx -
                     float4(Dleft, Dtop, Dright, Dbottom));
  float4 edges = step(threshold.xxxx, delta);

  if (dot(edges, 1.0) == 0.0) {
    discard;
  }

  return edges;
}
```

Listing 4.1. Edge detection shader.

4.3 Obtaining Blending Weights

In order to calculate the blending weights we first search for the distances to the ends of the line the edgel belongs to, using the edges texture obtained in the previous pass (see Section 4.3.1). Once these distances are known, we can use them to fetch the crossing edges at both ends of the line (see Section 4.3.2). These crossing edges indicate the type of pattern we are dealing with. The distances to the ends of the line and the type of pattern are used to access the precalculated texture (see Section 4.3.3) in which we store the areas that are used as blending weights for the final pass.

As mentioned before, to share calculations between adjacent pixels, we take advantage of the fact that two adjacent pixels share the same boundary, and

```
float4 BlendingWeightCalculationPS(
         float4 position: SV_POSITION,
         float2 texcoord: TEXCOORD0): SV_TARGET {
  float4 weights = 0.0;

  float2 e = edgesTex.SampleLevel(PointSampler,
                                  texcoord, 0).rg;

  [branch]
  if (e.g) { // Edge at north.
    float2 d = float2(SearchXLeft(texcoord),
                      SearchXRight(texcoord));

    // Instead of sampling between edges, we sample at -0.25,
    // to be able to discern what value each edgel has.
    float4 coords = mad(float4(d.x, -0.25, d.y + 1.0, -0.25),
                        PIXEL_SIZE.xyxy, texcoord.xyxy);
    float e1 = edgesTex.SampleLevel(LinearSampler,
                                    coords.xy, 0).r;
    float e2 = edgesTex.SampleLevel(LinearSampler,
                                    coords.zw, 0).r;
    weights.rg = Area(abs(d), e1, e2);
  }

  [branch]
  if (e.r) { // Edge at west.
    float2 d = float2(SearchYUp(texcoord),
                      SearchYDown(texcoord));

    float4 coords = mad(float4(-0.25, d.x, -0.25, d.y + 1.0),
                        PIXEL_SIZE.xyxy, texcoord.xyxy);
    float e1 = edgesTex.SampleLevel(LinearSampler,
                                    coords.xy, 0).g;
    float e2 = edgesTex.SampleLevel(LinearSampler,
                                    coords.zw, 0).g;
    weights.ba = Area(abs(d), e1, e2);
  }

  return weights;
}
```

Listing 4.2. Blending weights calculation shader.

we perform area calculation on a per-edgel basis. However, even though two adjacent pixels share the same calculation, the resulting a value is different for each of them: only one has a blending weight a, whereas for the opposite one, a equals zero (pixels $(1, 2)$ and $(1, 1)$, respectively, in Figure 4.2). The one exception to this is the case in which the pixel lies at the middle of a line of odd length (as pixel $(2, 1)$ in Figure 4.2); in this case both the actual pixel and its opposite have a nonzero value for a. As a consequence, the output of this pass is a texture which, for each pixel, stores the areas at each side of its corresponding edges (by *the areas at each side* we refer to those of the actual pixel and its opposite). This yields two values for north edges and two values for west edges in the final blending weights texture. Finally, the weights stored in this texture will be used in the third pass to perform the final blending. Listing 4.2 shows the source code of this pass; Figure 4.1 (center right) is the resulting image.

4.3.1 Searching for Distances

The search for distances to the ends of the line is performed using an iterative algorithm, which in each iteration checks whether the end of the line has been reached. To accelerate this search, we leverage the fact that the information stored in the edges texture is binary—it simply encodes whether an edgel exists—and query from positions between pixels using bilinear filtering for fetching two pixels at a time (see Figure 4.3). The result of the query can be: a) 0.0, which means that neither pixel contains an edgel, b) 1.0, which implies an edgel exists in both pixels, or c) 0.5, which is returned when just one of the two pixels contains an edgel. We stop the search if the returned value is lower than one.[2] By using a simple approach like this, we are introducing two sources of inaccuracy:

1. We do not stop the search when encountering an edgel perpendicular to the line we are following, but when the line comes to an end;

2. When the returned value is 0.5 we cannot distinguish which of the two pixels contains an edgel.

Although an error is introduced in some cases, it is unnoticeable in practice—the speed-up is considerable since it is possible to jump two pixels per iteration. Listing 4.3 shows one of the distance search functions.

In order to make the algorithm practical in a game environment, we limit the search to a certain distance. As expected, the greater the maximum length, the better the quality of the antialiasing. However, we have found that, for the majority of cases, distance values between 8 and 12 pixels give a good trade-off between quality and performance.

[2] In practice we use 0.9 due to bilinear filtering precision issues.

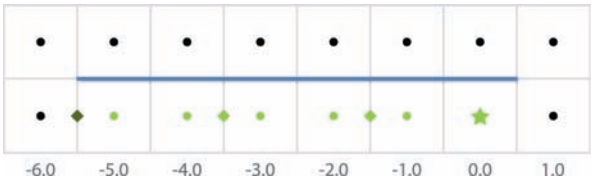


Figure 4.3. Hardware bilinear filtering is used when searching for distances from each pixel to the end of the line. The color of the dot at the center of each pixel represents the value of that pixel in the edges texture. In the case shown here, distance search of the left end of the line is performed for the pixel marked with a star. Positions where the edges texture is accessed, fetching pairs of pixels, are marked with rhombuses. This allows us to travel twice the distance with the same number of accesses.

In the particular case of the Xbox 360 implementation, we make use of the `tfetch2D` assembler instruction, which allows us to specify an offset in pixel units with respect to the original texture coordinates of the query. This instruction is limited to offsets of -8 and 7.5, which constrains the maximum distance that can be searched. When searching for distances greater than eight pixels, we cannot use the hardware as efficiently and the performance is affected negatively.

```
float SearchXLeft(float2 texcoord) {
  texcoord -= float2(1.5, 0.0) * PIXEL_SIZE;
  float e = 0.0;
  // We offset by 0.5 to sample between edges, thus fetching
  // two in a row.
  for (int i = 0; i < maxSearchSteps; i++) {
    e = edgesTex.SampleLevel(LinearSampler, texcoord, 0).g;
    // We compare with 0.9 to prevent bilinear access precision
    // problems.
    [flatten] if (e < 0.9) break;
    texcoord -= float2(2.0, 0.0) * PIXEL_SIZE;
  }
  // When we exit the loop without finding the end, we return
  // -2 * maxSearchSteps.
  return max(-2.0 * i - 2.0 * e, -2.0 * maxSearchSteps);
}
```

Listing 4.3. Distance search function (search in the left direction case).

Figure 4.4. Examples of the four possible types of crossing edge and corresponding value returned by the bilinear query of the edges texture. The color of the dot at the center of each pixel represents the value of that pixel in the edges texture. The rhombuses, at a distance of 0.25 from the center of the pixel, indicate the sampling position, while their color represents the value returned by the bilinear access.

4.3.2 Fetching Crossing Edges

Once the distances to the ends of the line are calculated, they are used to obtain the crossing edges. A naive approach for fetching the crossing edge of an end of a line would be to query two edges. A more efficient approach is to use bilinear filtering for fetching both edges at one time, in a manner similar to the way the distance search is done. However, in this case we must be able to distinguish the actual value of each edgel, so we query with an offset of 0.25, allowing us to distinguish which edgel is equal to 1.0 when only one of the edges is present. Figure 4.4 shows the crossing edge that corresponds to each of the different values returned by the bilinear query.

4.3.3 The Precomputed Area Texture

With distance and crossing edges information at hand, we now have all the required inputs to calculate the area corresponding to the current pixel. As this is an expensive operation, we opt to precompute it in a four-dimensional table which is stored in a conventional two-dimensional texture (see Figure 4.2 (right)).[3] This texture is divided into subtextures of size 9×9, each of them corresponding to a pattern type (codified by the fetched *crossing edges* $e1$ and $e2$ at each end of the line). Inside each of these subtextures, (u, v) coordinates correspond to distances to the ends of the line, eight being the maximum distance reachable. Resolution can be increased if a higher maximum distance is required. See Listing 4.4 for details on how the precomputed area texture is accessed.

To query the texture, we first convert the bilinear filtered values $e1$ and $e2$ to an integer value in the range 0..4. Value 2 (which would correspond to value 0.5 for $e1$ or $e2$) cannot occur in practice, which is why the corresponding row and column in the texture are empty. Maintaining those empty spaces in the texture

[3] The code to generate this texture is available in the web material.

```
#define NUM_DISTANCES 9
#define AREA_SIZE (NUM_DISTANCES * 5)

float2 Area(float2 distance, float e1, float e2) {
    // * By dividing by AREA_SIZE - 1.0 below we are
    //   implicitely offsetting to always fall inside a pixel.
    // * Rounding prevents bilinear access precision problems.
    float2 pixcoord = NUM_DISTANCES *
                      round(4.0 * float2(e1, e2)) + distance;
    float2 texcoord = pixcoord / (AREA_SIZE - 1.0);
    return areaTex.SampleLevel(PointSampler, texcoord, 0).rg;
}
```

Listing 4.4. Precomputed area texture access function.

allows for a simpler and faster indexing. The `round` instruction is used to avoid possible precision problems caused by the bilinear filtering.

Following the same reasoning (explained at the beginning of the section) by which we store area values for two adjacent pixels in the same pixel of the final blending weights texture, the precomputed area texture needs to be built on a per-edgel basis. Thus, each pixel of the texture stores two a values, one for a pixel and one for its opposite. (Again, a will be zero for one of them in all cases with the exception of those pixels centered on lines of odd length.)

4.4 Blending with the Four-Neighborhood

In this last pass, the final color of each pixel is obtained by blending the actual color with its four neighbors, according to the area values stored in the weights texture obtained in the previous pass. This is achieved by accessing three positions of the blending weights texture:

1. the current pixel, which gives us the north and west blending weights;
2. the pixel at the south;
3. the pixel at the east.

Once more, to exploit hardware capabilities, we use four bilinear filtered accesses to blend the current pixel with each of its four neighbors. Finally, as one pixel can belong to four different lines, we find an average of the contributing lines. Listing 4.5 shows the source code of this pass; Figure 4.1 (right) shows the resulting image.

```
float4 NeighborhoodBlendingPS(
          float4 position: SV_POSITION,
          float2 texcoord: TEXCOORD0): SV_TARGET {
  float4 topLeft = blendTex.SampleLevel(PointSampler,
                                        texcoord, 0);
  float right    = blendTex.SampleLevel(PointSampler,
                                        texcoord, 0,
                                        int2(0, 1)).g;
  float bottom   = blendTex.SampleLevel(PointSampler,
                                        texcoord, 0,
                                        int2(1, 0)).a;
  float4 a = float4(topLeft.r, right, topLeft.b, bottom);

  float sum = dot(a, 1.0);

  [branch]
  if (sum > 0.0) {
    float4 o = a * PIXEL_SIZE.yyxx;
    color = mad(colorTex.SampleLevel(LinearSampler,
                texcoord + float2( 0.0, -o.r), 0), a.r, color);
    color = mad(colorTex.SampleLevel(LinearSampler,
                texcoord + float2( 0.0,  o.g), 0), a.g, color);
    color = mad(colorTex.SampleLevel(LinearSampler,
                texcoord + float2(-o.b,  0.0), 0), a.b, color);
    color = mad(colorTex.SampleLevel(LinearSampler,
                texcoord + float2( o.a,  0.0), 0), a.a, color);
    return color / sum;
  } else {
    return colorTex.SampleLevel(LinearSampler, texcoord, 0);
  }
}
```

Listing 4.5. Four-neighborhood blending shader.

4.5 Results

Qualitywise, our algorithm lies between $4x$ and $8x$ MSAA, requiring a memory consumption of only $1.5x$ the size of the backbuffer on a PC and of $2x$ on Xbox 360.[4] Figure 4.5 shows a comparison between our algorithm, $8x$ MSAA, and no antialiasing at all on images from Unigine Heaven Benchmark. A limitation of our algorithm with respect to MSAA is the impossibility of recovering subpixel

[4]The increased memory cost in the Xbox 360 is due to the fact that two-channel render targets with 8-bit precision cannot be created in the framework we used for that platform, forcing the usage of a four-channel render target for storing the edges texture.

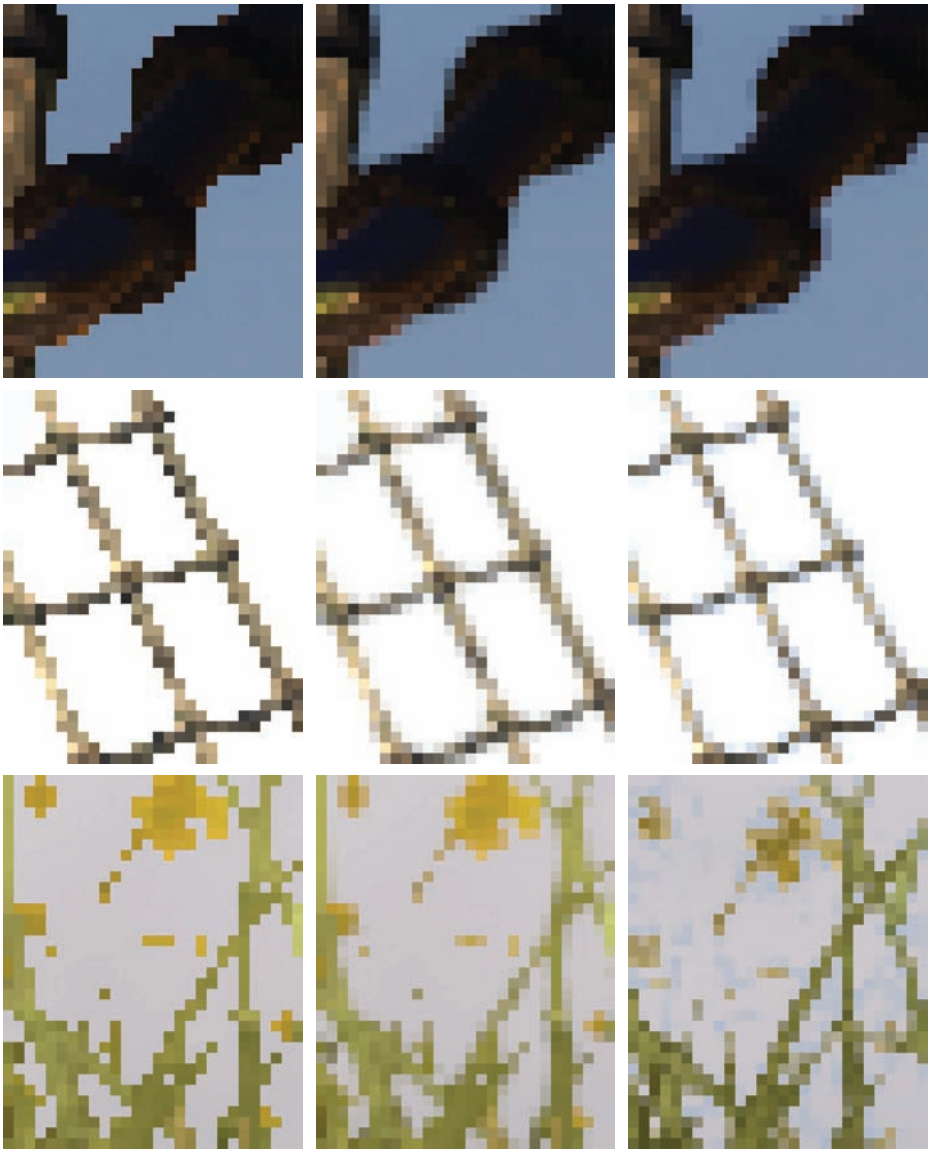


Figure 4.5. Examples of images without antialiasing, processed with our algorithm, and with $8x$ MSAA. Our algorithm offers similar results to those of $8x$ MSAA. A special case is the handling of alpha textures (bottom row). Note that in the grass shown here, alpha to coverage is used when MSAA is activated, which provides additional detail, hence the different look. As the scene is animated, there might be slight changes in appearance from one image to another. (Images from Unigine Heaven Benchmark courtesy of Unigine Corporation.)

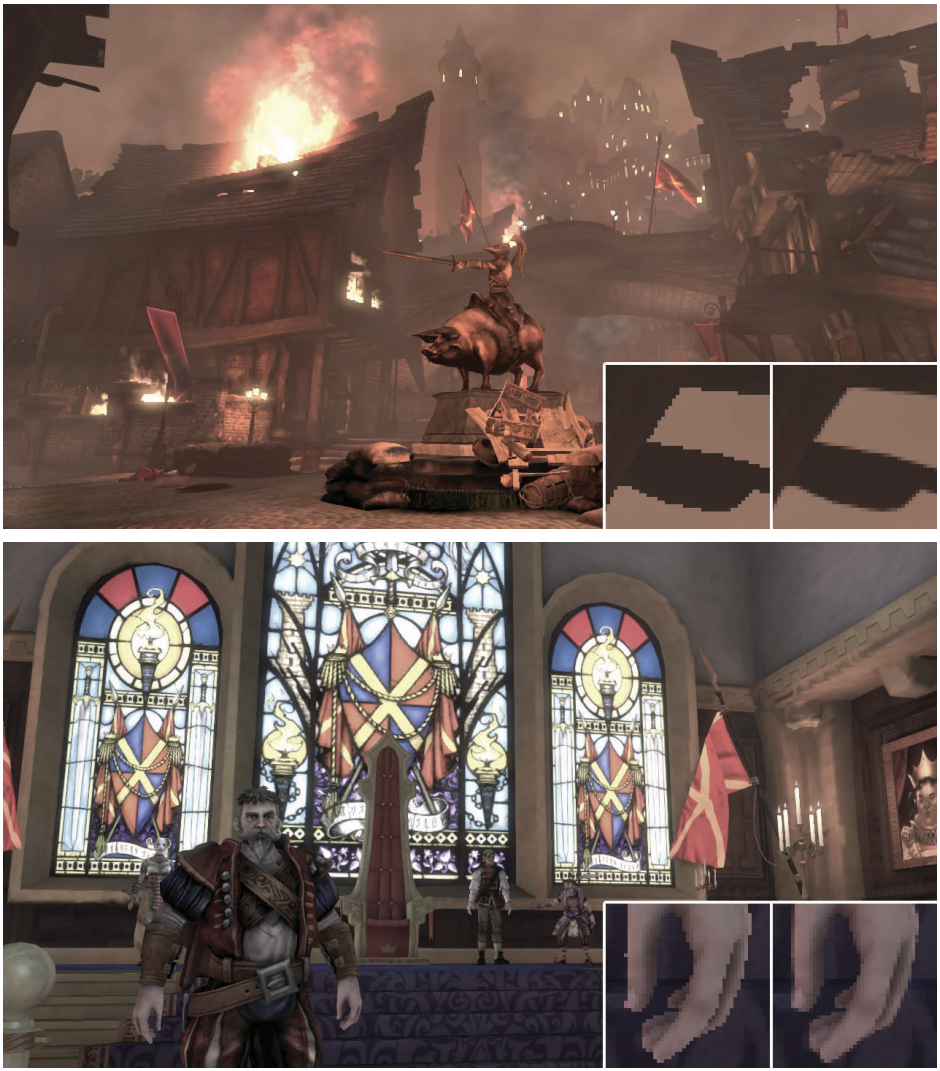


Figure 4.6. Images obtained with our algorithm. Insets show close-ups with no antialiasing at all (left) and processed with our technique (right). (Images from *Fable III* courtesy of Lionhead Studios.)

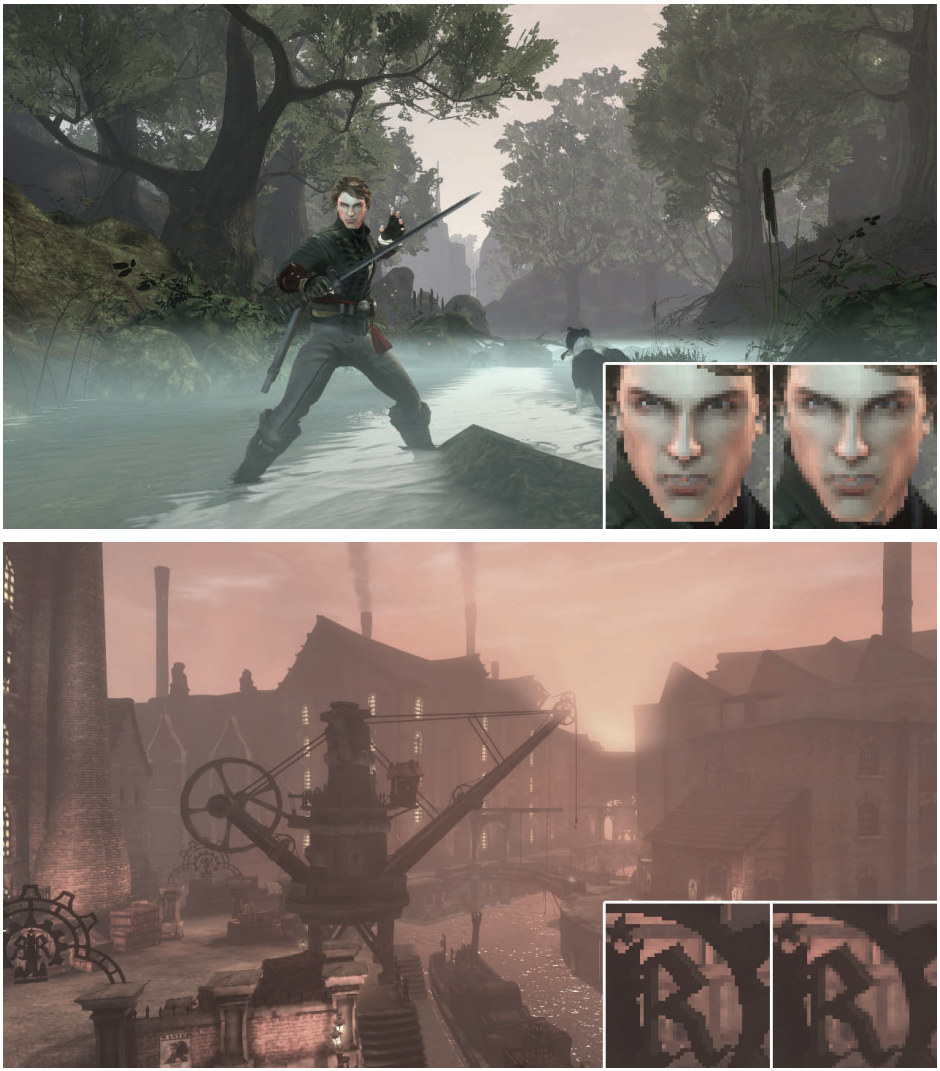


Figure 4.7. More images showing our technique in action. Insets show close-ups with no antialiasing at all (left) and processed with our technique (right). (Images from *Fable III* courtesy of Lionhead Studios.)

	Xbox 360		GeForce 9800 GTX+		
	Avg.	Std. Dev.	Avg.	Std. Dev.	Speed-up
Assassin's Creed	4.37 ms	0.61 ms	0.55 ms	0.13 ms	$6.31x^{\star}$
Bioshock	3.44 ms	0.09 ms	0.37 ms	0.00 ms	n/a
Crysis	3.92 ms	0.10 ms	0.44 ms	0.02 ms	$14.80x$
Dead Space	3.65 ms	0.45 ms	0.39 ms	0.03 ms	n/a
Devil May Cry 4	3.46 ms	0.34 ms	0.39 ms	0.04 ms	$5.75x$
GTA IV	4.11 ms	0.23 ms	0.47 ms	0.04 ms	n/a
Modern Warfare 2	4.38 ms	0.80 ms	0.57 ms	0.17 ms	$2.48x^{\star}$
NFS Shift	3.54 ms	0.35 ms	0.42 ms	0.04 ms	$14.84x$
Split/Second	3.85 ms	0.27 ms	0.46 ms	0.05 ms	n/a
S.T.A.L.K.E.R.	3.18 ms	0.05 ms	0.36 ms	0.01 ms	n/a
Grand Average	3.79 ms	0.33 ms	0.44 ms	0.05 ms	$11.80x$

Table 4.1. Average times and standard deviations for a set of well-known commercial games. A column showing the speed-up factor of our algorithm with respect to $8x$ MSAA is also included for the PC/DirectX 10 implementation. Values marked with * indicate $4x$ MSAA, since $8x$ was not available, and the grand average of these includes values only for $8x$ MSAA.

features. Further results of our technique, on images from *Fable III*, are shown in Figures 4.6 and 4.7. Results of our algorithm in-game are available in the web material.

As our algorithm works as a post-process, we have run it on a batch of screenshots of several commercial games in order to gain insight about its performance in different scenarios. Given the dependency of the edge detection on image content, processing times are variable. We have noticed that each game has a more or less unique "look-and-feel," so we have taken a representative sample of five screenshots per game. Screenshots were taken at 1280×720 as the typical case in the current generation of games. We used the slightly more expensive luminance-based edge detection, since we did not have access to depth information. Table 4.1 shows the average time and standard deviation of our algorithm on different games and platforms (Xbox 360/DirectX 9 and PC/DirectX 10), as well as the speed-up factor with respect to MSAA. On average, our method implies a speed-up factor of $11.80x$ with respect to $8x$ MSAA.

4.6 Discussion

This section includes a brief compilation of possible alternatives that we tried, in the hope that it would be useful for programmers employing this algorithm in the future.

Edges texture compression. This is perhaps the most obvious possible optimization, saving memory consumption and bandwidth. We tried two different alternatives: a) using 1 bit per edgel, and b) separating the algorithm into a vertical and a horizontal pass and storing the edges of four consecutive pixels in the RGBA

channels of each pixel of the edges texture (vertical and horizontal edges separately). This has two advantages: first, the texture uses less memory; second, the number of texture accesses is lower since several edges are fetched in each query. However, storing the values and—to a greater extent—querying them later, becomes much more complex and time consuming, given that bitwise operations are not available in all platforms. Nevertheless, the use of bitwise operations in conjunction with edges texture compression could further optimize our technique in platforms where they are available, such as DirectX 10.

Storing crossing edges in the edges texture. Instead of storing just the north and west edges of the actual pixel, we tried storing the crossing edges situated at the left and at the top of the pixel. The main reason for doing this was that we could spare one texture access when detecting patterns; but we realized that by using bilinear filtering we could also spare the access, without the need to store those additional edges. The other reason for storing the crossing edges was that, by doing so, when we searched for distances to the ends of the line, we could stop the search when we encountered a line perpendicular to the one we were following, which is an inaccuracy of our approach. However, the current solution yields similar results, requires less memory, and processing time is lower.

Two-pass implementation. As mentioned in Section 4.1, a two-pass implementation is also possible, joining the last two passes into a single pass. However, this would be more inefficient because of the repetition of calculations.

Storing distances instead of areas. Our first implementation calculated and stored only the distances to the ends of the line in the second pass, and they were then used in the final pass to calculate the corresponding blending weights. However, directly storing areas in the intermediate pass allows us to spare calculations, reducing execution time.

4.7 Conclusion

In this chapter, we have presented an algorithm crafted for the computation of antialiasing. Our method is based on three passes that detect edges, determine the position of each pixel inside those image features, and produce an antialiased result that selectively blends the pixel with its neighborhood according to its relative position within the line it belongs to. We also take advantage of hardware texture filtering, which allows us to reduce the number of texture fetches by half.

Our technique features execution times that make it usable in actual game environments, and that are far shorter than those needed for MSAA. The method presented has a minimal impact on existing rendering pipelines and is entirely implemented as an image post-process. Resulting images are between $4x$ and

$8x$ MSAA in quality, while requiring a fraction of their time and memory consumption. Furthermore, it can antialias transparent textures such as the ones used in alpha testing for rendering vegetation, whereas MSAA can smooth vegetation only when using alpha to coverage. Finally, when using luminance values to detect edges, our technique can also handle aliasing belonging to shading and specular highlights.

The method we are presenting solves most of the drawbacks of MSAA, which is currently the most widely used solution to the problem of aliasing; the processing time of our method is one order of magnitude below that of $8x$ MSAA. We believe that the quality of the images produced by our algorithm, its speed, efficiency, and pluggability, make it a good choice for rendering high quality images in today's game architectures, including platforms where benefiting from antialiasing, together with outstanding techniques like deferred shading, was difficult to achieve. In summary, we present an algorithm which challenges the current gold standard for solving the aliasing problem in real time.

4.8 Acknowledgments

Jorge would like to dedicate this work to his eternal and most loyal friend Kazán. The authors would like to thank the colleagues at the lab for their valuable comments, and Christopher Oat and Wolfgang Engel for their editing efforts and help in obtaining images. Thanks also to Lionhead Studios and Microsoft Games Studios for granting permission to use images from *Fable III*. We are very grateful for the support and useful suggestions provided by the Fable team during the production of this work. We would also like to express our gratitude to Unigine Corporation, and Denis Shergin in particular, for providing us with images and material for the video (available in the web material) from their Unigine Heaven Benchmark. This research has been funded by a Marie Curie grant from the 7^{th} Framework Programme (grant agreement no.: 251415), the Spanish Ministry of Science and Technology (TIN2010-21543) and the Gobierno de Aragón (projects OTRI 2009/0411 and CTPP05/09). Jorge Jimenez and Belen Masia are also funded by grants from the Gobierno de Aragón.

Bibliography

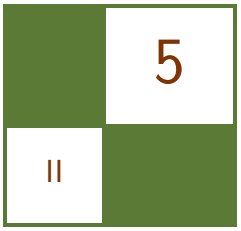
[Koonce 07] Rusty Koonce. "Deferred Shading in Tabula Rasa." In *GPU Gems 3*, pp. 429–457. Reading, MA: Addison Wesley, 2007.

[Reshetov 09] Alexander Reshetov. "Morphological Antialiasing." In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pp. 109–116. New York: ACM, 2009. Available online (http://visual-computing.intel-research.net/publications/papers/2009/mlaa/mlaa.pdf).

[Shishkovtsov 05] Oles Shishkovtsov. "Deferred Shading in S.T.A.L.K.E.R." In *GPU Gems 2*, pp. 143–166. Reading, MA: Addison Wesley, 2005.

[Sousa 07] Tiago Sousa. "Vegetation Procedural Animation and Shading in Crysis." In *GPU Gems 3*, pp. 373–385. Reading, MA: Addison Wesley, 2007.

[Thibieroz 09] Nicolas Thibieroz. "Deferred Shading with Multisampling Anti-Aliasing in DirectX 10." In *ShaderX*^{7}, pp. 225–242. Hingham, MA: Charles River Media, 2009.



Volume Decals

Emil Persson

5.1 Introduction

Decals are often implemented as textured quads that are placed on top of the scene geometry. While this implementation works well enough in many cases, it can also provide some challenges. Using decals as textured quads can cause Z-fighting problems. The underlying geometry may not be flat, causing the decal to cut into the geometry below it. The decal may also overhang an edge, completely ruining its effect. Dealing with this problem often involves clipping the decal to the geometry or discarding it entirely upon detecting the issue. Alternatively, very complex code is needed to properly wrap the decal around arbitrary meshes, and access to vertex data is required. On a PC this could mean that system-memory copies of geometry are needed to maintain good performance. Furthermore, disturbing discontinuities can occur, as in the typical case of shooting a rocket into a corner and finding that only one of the walls got a decal or that the decals do not match up across the corner. This article proposes a technique that overcomes all of these challenges by projecting a decal volume onto the underlying scene geometry, using the depth buffer.

5.2 Decals as Volumes

5.2.1 Finding the Scene Position

The idea behind this technique is to render the decal as a volume around the selected area. Any convex volume shape can be used, but typical cases would be spheres and boxes. The fragment shader computes the position of the underlying geometry by sampling the depth buffer. This can be accomplished as follows:

```
// texCoord is the pixel's normalized screen position
float depth = DepthTex.Sample(Filter, texCoord);
float4 scrPos = float4(texCoord, depth, 1.0f);
float4 wPos = mul(scrPos, ScreenToWorld);
```

Figure 5.1. Example decal rendering.

```
float3 pos = wPos.xyz / wPos.w;
// pos now contains pixel position in world space
```

The `ScreenToWorld` matrix is a composite matrix of two transformations: namely the transformation from screen coordinates to clip space and then from clip space to world space. Transforming from world space to clip space is done with the regular `ViewProjection` matrix, so transforming in the other direction is done with the inverse of this matrix. Clip space ranges from -1 to 1 in x and y, whereas the provided texture coordinates are in the range of 0 to 1, so we also need an initial scale-bias operation baked into the matrix. The matrix construction code could look something like this:

```
float4 ScaleToWorld = Scale(2, -2, 1) *
    Translate(-1, 1, 0) * Inverse(ViewProj);
```

What we are really interested in, though, is the local position relative to the decal volume. The local position is used as a texture coordinate used to sample a volume texture containing a volumetric decal (see Figure 5.1). Since the decal is a volumetric texture, it properly wraps around nontrivial geometry with no discontinuities (see Figure 5.2). To give each decal a unique appearance, a random rotation can also be baked into the matrix for each decal. Since we do a matrix transformation we do not need to change the shader code other than to name the matrix more appropriately as `ScreenToLocal`, which is then constructed as follows:

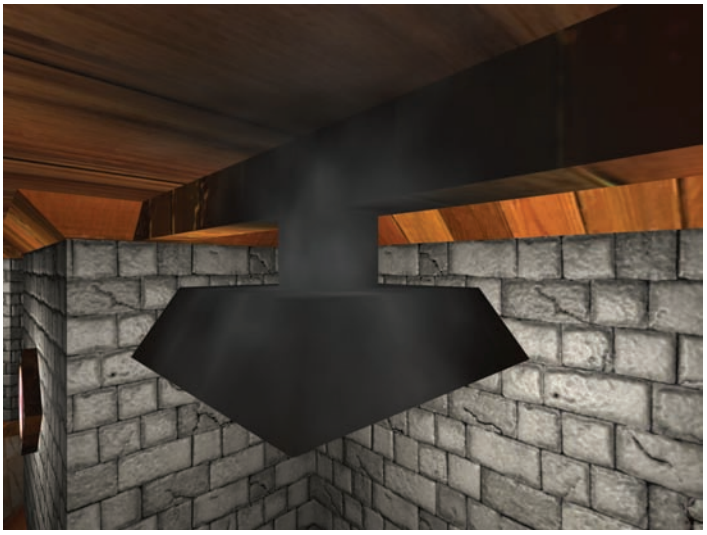


Figure 5.2. Proper decal wrapping around nontrivial geometry.

```
float4 ScreenToLocal = Scale(2, -2, 1) *
    Translate(-1, 1, 0) * Inverse(ViewProj) *
    DecalTranslation * DecalScale * DecalRotation;
```

The full fragment shader for this technique is listed below and a sample with full source code is available in the web materials.

```
Texture2D <float> DepthTex;
SamplerState DepthFilter;

Texture3D <float4> DecalTex;
SamplerState DecalFilter;

cbuffer Constants
{
    float4x4 ScreenToLocal;
    float2 PixelSize;
};

float4 main(PsIn In) : SV_Target
{

    // Compute normalized screen position
    float2 texCoord = In.Position.xy * PixelSize;
```

```
    // Compute local position of scene geometry
    float depth = DepthTex.Sample(DepthFilter, texCoord);
    float4 scrPos = float4(texCoord, depth, 1.0f);
    float4 wPos = mul(scrPos, ScreenToLocal);

    // Sample decal
    float3 coord = wPos.xyz / wPos.w;
    return DecalTex.Sample(DecalFilter, coord);
}
```

Listing 5.1. The full fragment shader.

5.2.2 Implementation and Issues

In a deferred-rendering system [Thibieroz 04] this technique fits perfectly. The decals can be applied after the *geometry buffer* (G-buffer) pass and the relevant attributes, such as diffuse color and specularity, can simply be updated, and then lighting can be applied as usual. This technique also works well with a light pre-pass renderer [Engel 09], in which case lighting information is readily available for use in the decal pass.

In a forward rendering system the decals will be applied after lighting. In many cases this is effective also, for instance, for burn marks after explosions, in which case the decals can simply be modulated with the destination buffer. With more complicated situations, such as blending with alpha, as is typically the case for bullet holes, for instance, the decal application may have to take lighting into account. One solution is to store the overall lighting brightness into alpha while rendering the scene; the decal can then pre-multiply source color with alpha in the shader and multiply with destination alpha in the blender to get reasonable lighting. This will not take light color into account, but may look reasonable if lighting generally is fairly white. Another solution is to simply go by the attenuation of the closest light and not take any normal into account. Alternatively, a normal can be computed from the depth buffer, although this is typically slow and has issues of robustness [Persson 09].

One issue with this technique is that it applies the decal on everything within the decal volume. This is not a problem for static objects, but if you have a large decal volume and dynamic objects move into it they will get the decal smeared onto them, for instance, if you previously blew a bomb in the middle of the road and a car is passing through at a later time. This problem can be solved by drawing dynamic objects after the decal pass. A more elaborate solution is to render decals and dynamic objects in chronological order so that objects that are moved after the decal is added to the scene will not be affected by the decal. This will allow dynamic objects to be affected by decals as well. Another solution is to use object IDs. The decal can store the IDs of objects that intersected the decal

volume at the time it was added to the scene and cull for discarded pixels that do not belong to any of those objects.

5.2.3 Optimizations

On platforms where the depth-bounds test is supported, the depth-bounds test can be used to improve performance. On other platforms, dynamic branching can be used to emulate this functionality by comparing the sample depth to the depth bounds. However, given that the shader is relatively short and typically a fairly large number of fragments survive the test, it is recommended to benchmark to verify that it actually improves performance. In some cases it may in fact be faster to not attempt to cull anything.

5.2.4 Variations

In some cases it is desirable to use a two-dimensional texture instead of a volume decal. Volume textures are difficult to author and consume more memory. Not all cases translate well from a two-dimensional case to three dimensions. A bullet hole decal can be swept around to a spherical shape in the three-dimensional case and can then be used in any orientation, but this is not possible for many kinds of decals; an obvious example is a decal containing text, such as a logo or graffiti tag.

An alternate technique is to sample a two-dimensional texture using just the x, y components of the final coordinates. The z component can be used for fading. When a volume texture is used, you can get an automatic fade in all directions by letting the texture alpha fade to zero toward the edges and using a border color with an alpha of zero. In the 2D case you will have to handle the z direction yourself.

Two-dimensional decals are not rotation invariant so when placing them in the scene they must be oriented such that they are projected sensibly over the underlying geometry. The simplest approach would be to just align the decal plane with the normal of the geometry at the decal's center point. Some problematic cases exist though, such as when wrapping over a corner of a wall. If it is placed flat against the wall you will get a perpendicular projection on the other side of the corner with undesirable texture-stretching as a result.

An interesting use of the two-dimensional case is to simulate a blast in a certain direction. This can be accomplished by using a pyramid or frustum shape from the point of the blast. When the game hero shoots a monster you place a frustum from the bullet-impact point on the monster to the wall behind it in the direction of the bullet and you will get the effect of blood and slime smearing onto the wall. The projection matrix of this frustum will have to be baked into the `ScreenToLocal` matrix to get the proper projection of the texture coordinates.

The blast technique can also be varied for a cube decal scenario. This would better simulate the effect of a grenade blast. In this case a cube or sphere would be rendered around the site of the blast and a cubemap lookup is performed with the final coordinates. Fading can be effected using the length of the coordinate vector.

To improve the blast effect you can use the normals of underlying geometry to eliminate the decal on back-facing geometry. For the best results, a shadowmapesque technique can be used to make sure only the surfaces closest to the front get smeared with the decal. This "blast-shadow map" typically has to be generated only once at the time of the blast and can then be used for the rest of the life of the decal. Using the blast-shadow map can ensure splatter happens only in the blast shadow of monsters and other explodable figures, whereas areas in the blast-shadow map that contain static geometry only get scorched. This requires storing a tag in the shadow buffer for pixels belonging to monsters, however. Creative use of the shadow map information also can be used to vary the blood-splatter intensity over the distance from the blast to the monster and from the monster to the smeared wall.

5.3 Conclusions

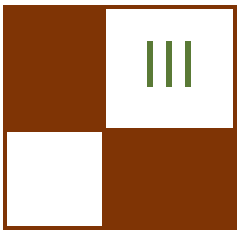
An alternate approach for decal rendering has been shown that suggests solutions to many problems of traditional decal-rendering techniques. Using volumes instead of flat decal geometry allows for continal decals across nontrivial geometry. It also eliminates potentially expensive buffer locks or the need for system-memory buffer copies.

Bibliography

[Thibieroz 04] Nicolas Thibieroz. "Deferred Shading with Multiple Render Targets." In *ShaderX^2: Shader Programming Tips and Tricks with DirectX 9*, edited by Wolfgang Engel, pp. 251–269. Plano, TX: Wordware Publishing, 2004.

[Engel 09] Wolfgang Engel. "Designing a Renderer for Multiple Lights - The Light Pre-Pass Renderer." In *ShaderX^7: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 655–666. Hingham, MA: Charles River Media, 2009.

[Persson 09] Emil Persson. "Making It Large, Beautiful, Fast and Consistent: Lessons Learned Developing Just Cause 2." in *GPU Pro: Advanced Rendering Techniques*, pp. 571–596. Natick, MA: A K Peters, 2010.



III Global Illumination Effects

The good news: global illumination effects (exceeding ambient occlusion) have found their way into production! The advances in graphics hardware capabilities in recent years, combined with smart algorithms and phenomenologically well-motivated—and also well-understood—approximations of light transport, allow the rendering of ever-increasing numbers of phenomena in real time. This section includes four articles describing rendering techniques of global illumination effects, and all of them are suited for direct rendering applications in real time.

Reprojection caching techniques, introduced about three years ago, can exploit temporal coherence in rendering and, by this, reduce computation for costly pixel shaders by reusing results from previous frames. In "Temporal Coherence to Improve Screen-Space Ambient Occlusion," Oliver Mattausch, Daniel Scherzer, and Michael Wimmer adapt temporal coherence for improving the performance of *screen-space ambient occlusion* (SSAO) techniques. Their algorithm reuses *ambient occlusion* (AO) sample information from previous frames if available, and adaptively generates more AO samples as needed. Spatial filtering is applied only to regions where the AO computation does not yet converge. This improves the overall quality as well as performance of SSAO.

In "Level-of-Detail and Streaming Optimized Irradiance Normal Mapping," Ralf Habel, Anders Nilsson, and Michael Wimmer describe a clever technique for irradiance normal mapping, which has been successfully used in various games. They introduce a modified hemispherical basis (hierarchical, in the spirit of spherical harmonics) to represent low-frequency directional irradiance. The key to this basis is that it contains the traditional light map as one of its coefficients, and further basis functions provide additional directional information. This enables shader *level-of-detail* (LOD) (in which the light map is the lowest LOD), and streaming of irradiance textures.

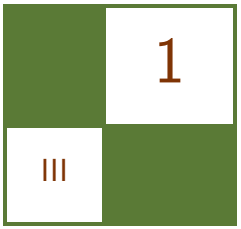
"Real-Time One-Bounce Indirect Illumination and Indirect Shadows Using Ray Tracing," by Holger Gruen, describes an easy-to-implement technique to achieve one-bounce indirect illumination, including shadowing of indirect light, in real time, which is often neglected for fully dynamic scenes. His method consists of three phases: rendering of indirect light with *reflective shadow maps* (RSMs), creating a three-dimensional grid as acceleration structure for ray-triangle intersection using the capabilities of Direct3D 11 hardware, and finally computing

the blocked light using RSMs and ray casting, which is then subtracted from the result of the first phase.

In their article, "Real-Time Approximation of Light Transport in Translucent Homogenous Media," Colin Barré-Brisebois and Marc Bouchard describe an amazingly simple method to render plausible translucency effects for a wide range of objects made of homogeneous materials. Their technique combines precomputed, screen-space thickness of objects with local surface variation into a shader requiring only very few instructions and running in real time on a PC and console hardware. The authors also discuss scalability issues and the artist friendliness of their shading technique.

"Real-Time Diffuse Global Illumination with Temporally Coherent Light Propagation Volumes," by Anton Kaplanyan, Wolfgang Engel, and Carsten Dachsbacher, describes the global-illumination approach used in the upcoming game *Crysis 2*. The technique consists of four stages: in the first stage all lit surfaces of the scene are rendered into RSMs. Then a sparse three-dimensional grid of radiance distribution is initialized with the generated *surfels* from the first stage. In the next step, the authors propagate the light in this grid using an iterative propagation scheme and, in the last stage, the resulting grid is used to illuminate the scene similarly to the irradiance volumes technique described by Natalya Tatarchuk in the article "Irradiance Volumes for Games."

—Carsten Dachsbacher



Temporal Screen-Space Ambient Occlusion

Oliver Mattausch, Daniel Scherzer, and Michael Wimmer

1.1 Introduction

Ambient occlusion (AO) is a shading technique that computes how much of the hemisphere around a surface point is blocked, and modulates the surface color accordingly. It is heavily used in production and real-time rendering, because it produces plausible global-illumination effects with relatively low computational cost. Recently it became feasible to compute AO in real time, mostly in the form of *screen-space ambient occlusion* (SSAO). SSAO techniques use the depth buffer as a discrete scene approximation, thus have a constant overhead and are simple to implement.

However, to keep the computation feasible in real time, concessions have to be made regarding the quality of the SSAO solution, and the SSAO evaluation has to be restricted to a relatively low number of samples. Therefore, the generated AO is usually prone to surface noise, which can be reduced in a post-processing step with a discontinuity filter. Depending on the chosen filter settings, we can either keep sharp features and accept some noise, or get a smooth but blurry solution due to filtering over the edges (as can be seen in Figure 1.1). Also, for dynamically moving objects, the noise patterns will sometimes appear to float on the surfaces, which is a rather distracting effect. To get a solution that is neither noisy nor blurry, many more samples have to be used. This is where *temporal coherence* comes into play.

1.1.1 Exploiting Temporal Coherence

The *reverse reprojection* technique [Scherzer et al. 07,Nehab et al. 07] allows us to reuse pixels from previous frames and refine them over time. This allows keeping

Figure 1.1. SSAO without temporal coherence (23 FPS) with 32 samples per pixel, with (a) a weak blur, (b) a strong blur. (c) TSSAO (45 FPS), using 8–32 samples per pixel (initially 32, 8 in a converged state). (d) Reference solution using 480 samples per frame (2.5 FPS). All images at 1024×768 resolution and using 32-bit precision render targets. The scene has 7 M vertices and runs at 62 FPS without SSAO.

the number of samples that are computed in a single frame low, while effectively accumulating hundreds of samples in a short amount of time. Note that ambient occlusion has many beneficial properties that make it well suited for temporal coherence: there is no directional dependence on the light source or the viewer, AO techniques consider only the geometry in a local neighborhood, and only the SSAO in a *pixel neighborhood* is affected by a change in the scene configuration.

In this article, we focus specifically on how to use reverse reprojection to improve the quality of SSAO techniques in a deferred shading pipeline. In particular, we show how to detect and handle changes to the SSAO caused by moving entities, animated characters, and deformable objects. We demonstrate that these cases, which are notoriously difficult for temporal coherence methods, can be significantly improved as well. A comparison of our *temporal SSAO* (TSSAO) technique with conventional SSAO and a reference solution in a static scene configuration can be seen in Figure 1.1.

Note that this algorithm is complementary to the method described in the "Fast Soft Shadows With Temporal Coherence" chapter of this book, which also provides code fragments that describe the reprojection process.

1.2 Ambient Occlusion

From a physical point of view, AO can be seen as the diffuse illumination due to the sky [Landis 02]. AO of a surface point p with normal n_p is computed as [Cook and Torrance 82]:

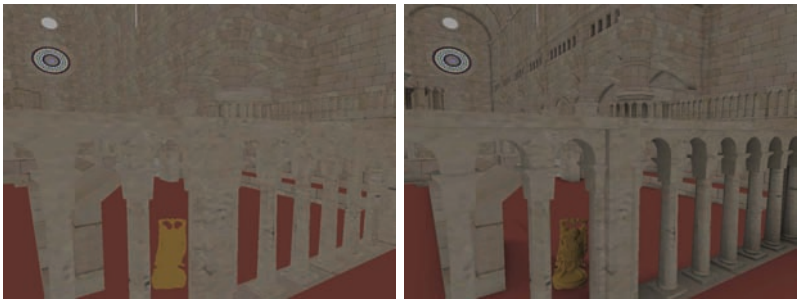


Figure 1.2. This figure compares rendering without (left) and with (right) AO, and shows that AO allows much better depth perception and feature recognition, without requiring any additional lighting.

$$\text{ao}(p, n_p) = \frac{1}{\pi} \int_{\Omega} V(p, \omega) D(|p - \xi|) n_p \cdot \omega d\omega, \tag{1.1}$$

where ω denotes all directions on the hemisphere and V is the (inverse) binary visibility function, with $V(p, \omega) = 1$ if the visibility in this direction is blocked by an obstacle, 0 otherwise. D is a monotonic decreasing function between 1 and 0 of the distance from p to ξ, the intersection point with the nearest surface. In the simplest case, D is a step function, considering obstacles within a certain sampling radius only, although a smooth falloff provides better results, (e.g., as given by an $exp(.)$ function).

Figure 1.2 demonstrates the visual impact of SSAO for the depth perception of a scene.

1.2.1 Screen Space Ambient Occlusion

SSAO methods attempt to approximate the original AO integral in screen space. Several versions of SSAO with different assumptions and trade-offs have been described. Note that our algorithm can be seen as a general strategy to improve an underlying SSAO method.

We assume that any SSAO method can be written as an average over contributions C which depend on a series of samples s_i:

$$AO_n(p) = \frac{1}{n} \sum_{i=1}^{n} C(p, s_i). \tag{1.2}$$

In order to approximate Equation (1.1) using the Monte Carlo integration, the contribution function for SSAO is often chosen as [Ritschel et al. 09]:

$$C(p, s_i) = V(p, s_i) D(|s_i - p|) \max(\cos(s_i - p, n_p), 0). \tag{1.3}$$

In contrast to Equation (1.1), directions have been substituted by actual sample points around p, and thus $V(p, s_i)$ is now a binary visibility function that gives 0 if s_i is visible from p, and 1 otherwise. The method is called "screen-space" ambient occlusion because V is evaluated by checking whether s_i is visible in screen space with respect to the current z-buffer. Note that some SSAO methods omit the depth test, and the contribution of s_i depends on distance and incident angle [Fox and Compton 08].

We assume that the samples s_i have been precomputed and stored in a texture, for example, a set of three-dimensional points uniformly distributed in the hemisphere, which are transformed into the tangent space of p for the evaluation of C. If this step is omitted and samples on the whole sphere are taken, then the sample contribution has to be doubled to account for wasted samples that are not on the upper hemisphere. $D(.)$ is a function of the distance to the sample point that can be used to modulate the falloff.

1.3 Reverse Reprojection

Reverse reprojection associates image pixels from the current frame with the pixels from the previous frame that represent the same world-space position. The technique allows the reuse of pixel content from previous frames for the current frame. The technique was shown to be useful for a variety of applications, like shadow mapping, antialiasing, or even motion blur [Rosado 07].

Reprojection techniques use two render targets in ping-pong fashion, one for the current frame and one representing the cached information from the previous frames. In our context we cache AO values and therefore denote this buffer as the *ambient-occlusion buffer*.

For static geometry, reprojection is constant for the whole frame, and can be carried out in the pixel shader or in a separate shading pass (in the case of deferred shading), using the view (V) and projection (P) matrices from the previous frame $f-1$ and the current frame f, where t denotes the post-perspective position of a pixel [Scherzer et al. 07]:

$$t_{f-1} = P_{f-1} V_{f-1} V_f^{-1} P_f^{-1} t_f. \tag{1.4}$$

In our deferred shading pipeline, we store *eye-linear* depth values for the current frame and the previous frame, and use them to reconstruct the world-space positions p. In our implementation, because we already store the world-space positions, we have only to transform the current world-space position p_f with the previous view-projection matrix $P_{f-1}V_{f-1}$ to get t_{f-1}. From t_{f-1} we calculate the correct lookup coordinates tex_{f-1} into the AO buffer by applying the perspective division and scaling the result to the range [0..1] (i.e., $tex_{f-1} = \frac{t_{f-1}+1}{2}$.).

1.3.1 Dynamic Objects

For dynamic scenes, the simple Equation (1.4) does not work, because reprojection depends on the transformations of moving objects. Therefore, it was proposed to apply the complete vertex transformation twice, once using the current transformation parameters (modeling matrix, skinning, etc.), and once using the parameters of the previous frame [Nehab et al. 07].

In a deferred shading pipeline, the previous position p_f needs to be accessed in a separate shading pass, where information about transformation parameters is already lost. Therefore, we store the *3D optical flow* $p_{f-1} - p_f$ in the frame buffer as another shading parameter (alongside normal, material, etc.), using a lower precision for these offset values than for the absolute depth values (16 bit instead of 32 bit).

1.4 Our Algorithm

In this section, we first describe the SSAO sample accumulation, then the detection and handling of pixels that carry invalid information, and last, some optimizations to our algorithm.

1.4.1 Refining the SSAO Solution Over Time

The main concept of our algorithm is to spread the computation of AO (Equation (1.2)) over several frames by using reprojection. Whenever possible, we take the solution from a previous frame that corresponds to an image pixel and refine it with the contribution of new samples computed in the current frame. In frame f, we calculate a new contribution C_f from k new samples:

$$C_f(p) = \frac{1}{k} \sum_{i=j_f(p)+1}^{j_f(p)+k} C(p, s_i), \tag{1.5}$$

where $j_f(p)$ counts the number of unique samples that have already been used in this solution. We combine the new solution with the previously computed solution:

$$\begin{aligned} AO_f(p) &= \frac{w_{f-1}(p_{f-1}) AO_{f-1}(p_{f-1}) + kC_f(p)}{w_{f-1}(p) + k}, \\ w_f(p) &= \min(w_{f-1}(p_{f-1}) + k, w_{\max}), \end{aligned} \tag{1.6}$$

where the weight w_{f-1} is the number of samples that have already been accumulated in the solution, or a predefined maximum after convergence has been reached.

Theoretically, this approach can use arbitrarily many samples. In practice, however, this is not advisable: since reprojection is not exact and requires bilinear filtering for reconstruction, each reprojection step introduces an error that exacerbates over time. This error is noticeable as an increasing amount of blur. Furthermore, the influence of newly computed samples becomes close to zero, and previously computed samples never get replaced. Therefore we clamp w_f to a user-defined threshold $w_{\max}$, which causes the influence of older contributions to decay over time. Thus,

$$\text{conv}(p) = w_f(p)/w_{\max} \tag{1.7}$$

is an indicator of the state of convergence. Note that for $w_{max} \to \infty$, accumulating contributions from Equation (1.5) correspond to $n \to \infty$ in Equation (1.2) and would thus converge to the correct value of the desired integration, except for the blurring discussed above.

Implementation notes. The value w_{f-1} is stored in a separate channel in the ambient occlusion buffer. In order to achieve fast convergence, we use a Halton sequence, which is known for its low discrepancy [Wang and Hickernell 00], for sample generation. As a starting index into this sample set, we use j_f, which we also store in the AO-buffer. In summary, the RGBA target of an AO-buffer stores the following parameters:

- the SSAO solution $C_{f-1}(p)$
- the weight of the previous solution w_{f-1}
- the starting index j_{f-1}
- the eye-linear depth d_{f-1}.

The current index position is propagated to the next frame by means of reverse reprojection as with the SSAO values. In order to prevent the index position from being interpolated by the hardware and introducing a bias into the sequence, it is important to always fetch the index value from the nearest pixel center in the AO-buffer. The pixel center can be found using the formula in Equation (1.8), given the reprojected coordinates tex_{f-1}:

$$\text{pixelcenter} = \frac{\lfloor \text{tex}_{f-1}\text{res}_{x,y} \rfloor + 0.5}{\text{res}_{x,y}}, \tag{1.8}$$

where $\text{res}_{x,y}$ is the current frame buffer resolution in x and y.

1.4.2 Detecting and Dealing with Invalid Pixels

When reprojecting a fragment, we need to check whether the pixel looked up in the previous frame actually corresponds to the current pixel, i.e., whether the cached AO value is valid. If it became invalid, we have to invalidate the previous solution accordingly. In order to detect such invalid pixels, we check if any one of the following three conditions has occurred: 1.) a disocclusion of the current fragment [Scherzer et al. 07], 2.) changes in the *sample neighborhood* of the fragment, and 3.) a fragment that was previously outside the frame buffer. Following, we discuss these cases.

Detecting disocclusions. We check for disocclusions by comparing the current depth of the fragment d_f to the depth of the cached value at the reprojected fragment position d_{f-1}. In particular, we compare the *relative* depth differences of the eye-linear depth values:

$$|1 - \frac{d_f}{d_{f-1}}| < \epsilon. \tag{1.9}$$

Equation (1.9) gives stable results for large scenes with a wide depth range, which are not oversensitive at the near plane and are sufficiently sensitive when approaching the far-plane regions. In case of a disocclusion, we always discard the previous solution by resetting w_{f-1} to 0 and we compute a completely new AO solution.

Detecting changes in the neighborhood. Testing for disocclusions may be sufficient for avoiding most temporal-coherence artifacts for methods that affect only the current pixel, like super sampling, or for SSAO in a purely static scene. However, shading methods like SSAO gather information from neighboring pixels using a spatial sampling kernel. Hence, in dynamic environments, we have to take into account that the shading of the current pixel can be affected by nearby moving objects, even if no disocclusion of the pixel itself has happened. Consider, for

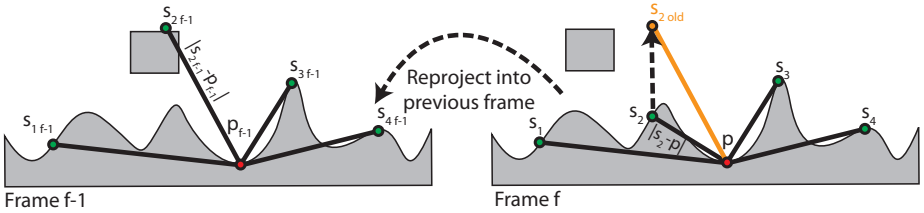


Figure 1.3. The distance of p to sample point s_2 in the current frame (right) differs significantly from the distance of p_{f-1} to $s_{2\,f-1}$ in the previous frame (left), so we assume that a local change of geometry occurred, which affects the shading of P.

example, a scenario wherein a box is lifted from the floor. The SSAO values of pixels in the contact-shadow area surrounding the box change, even if there is no disocclusion of the pixel itself.

The size of the neighborhood to be checked is equivalent to the size of the sampling kernel used for SSAO. Checking the complete neighborhood of a pixel would be prohibitively expensive, and therefore we use sampling. Actually, it turns out that we already have a set of samples, namely the ones used for AO generation. That means that we effectively use our AO sampling kernel for two purposes: for computing the current contribution $C_f(p)$, and to test for validity.

Our invalidation scheme is visualized in Figure 1.3. The validity of a sample s_i for shading a pixel p can be estimated by computing the change in relative positions of sample and pixel:

$$\delta(s_i) = ||s_i - p| - |s_{if-1} - p_{f-1}||. \tag{1.10}$$

The reprojected position s_{if-1} is computed from the offset vector stored for s_i (recall that the first rendering pass stores the offset vectors for all pixels in the frame buffer for later access by the SSAO-shading pass). Note that, for the neighborhood test, we use only those samples that lie in front of the tangent plane of p, since only those samples actually modify the shadow term.

Theoretically we could also check if the angle between surface normal and vector to the sample point has changed by a significant amount from one frame to the next, and practical cases are imaginable when the vector length is not enough. However, this would require more information to be stored (the surface normal of every pixel in the previous frame), and in all our tests we found it sufficient to evaluate Equation (1.10).

Note that in order to avoid one costly texture lookup when fetching p_f, the required values for this test and for the AO computation should be stored in a single render target.

Smooth invalidation. It makes perfect sense to use a binary threshold to detect disocclusions. In this spirit, for the neighborhood check of a pixel we could evaluate $\delta(s_i) < \epsilon$ for all samples and discard the previous solution for this pixel if this condition is violated for any of the samples. However, consider, for example, a slowly deforming surface, where the AO will also change slowly. In such a case it is not necessary to fully discard the previous solution. Instead we introduce a new continuous definition of invalidation that takes a measure of change into account. This measure of change is given by $\delta(s_i)$ at validation sample position s_i, as defined in Equation (1.10). In particular, we compute a *confidence* value $\mathrm{conf}(s_i)$ between 0 and 1. It expresses the degree to which the previous SSAO solution is still valid:

$$\mathrm{conf}(s_i) = 1 - \frac{1}{1 + S\delta(s_i)}.$$

(a) Confidence evaluation (b) Confidence values

Figure 1.4. (a) Confidence function depending on the distance difference δ for smoothing factor $S = 5, 15, 30, 50$. (b) Visualization of the confidence values computed by our smooth invalidation technique, showing a rotation (left), a translation (middle), and an animated (walking) character (right). We use a continuous scale from red (confidence=0) to white (confidence=1).

The parameter S controls the smoothness of the invalidation, and is set to a value ($15 \leq S \leq 30$) in our current implementation. As can be seen in Figure 1.4, for different values of S, the confidence is 1 if the relative distance has not changed ($\delta(x) = 0$), and approaches 0 for large values of $\delta(s_i)$. The overall confidence of the previous AO solution is given by

$$conf(p) = \min(conf(s_0), .., conf(s_k)).$$

We multiply it with w_t to modify the weight of the old solution in Equation (1.6). Also, in order to prevent flickering artifacts in regions with large changes, we do not increase the index into the array of samples if the convergence is smaller than a threshold (e.g., for $conv(p) < 0.5$) to reuse the same samples.

Figure 1.5 shows the effect of our novel invalidation scheme on a scene with a translational movement. Checking only disocclusions causes artifacts visible

Figure 1.5. Rotating dragon model using different values for smooth invalidation factor S. (left) $S = 0$ (i.e., no invalidation), (middle) $S = 100$, (right) $S = 15$. Note that no invalidation causes a wrong shadow (left), while a too high value causes unwanted noise in the shadow (middle).

as wrong contact shadows (left). Additionally, checking the pixel neighborhood for changes in the AO using our smooth invalidation technique allows for correct shadows (middle and right). Choosing a too high value for S will remove the temporal-coherence artifacts, but produces too much noise (middle). On the other hand, there is much less noise at the transitions between the silhouettes of moving objects and the background when choosing a proper value for S (right).

Handling of frame-buffer borders. Samples that fall outside of the frame buffer carry incorrect information that should not be propagated. Hence we check for each pixel if one or more of the samples have been outside the frame buffer in the *previous frame*. In this case, we do not use smooth invalidation, but discard the previous values completely since they are undefined. In the same spirit, we do not use samples that fall outside of the frame buffer to compute our confidence values.

1.4.3 Adaptive Convergence-Aware Spatial Filter

SSAO methods usually apply a spatial-filtering pass after shading computations in order to prevent noise artifacts caused by insufficient sampling rates. We also apply spatial filtering, but only as long as the temporal coherence is not sufficient. Variants of the cross bilateral filter [Eisemann and Durand 04] are typically used, where filtering over edges is avoided by taking the depth differences into account. Although this filter is not formally separable, in a real-time setting it is usually applied separately in x and y directions to make evaluation feasible.

In contrast to previous approaches, we have additional information for this filter which can greatly reduce noise (i.e., the convergence $\mathrm{conv}(p)$ of our AO values in pixel p (Equation 1.7)). Recently disoccluded pixels (e.g., in a thin silhouette region) can gather more information from nearby converged pixels than from other unreliable pixels. Furthermore, we apply the filter kernel directly to world-space distances. This application automatically takes depth differences into account, and can detect discontinuities in cases of high depth differences:

$$\begin{aligned} AO_{\mathrm{filt}}(p) &= \frac{1}{K(p)} \sum_{x \in F} g(|p-x|)\mathrm{conv}(x)AO(x), \\ K(p) &= \sum_{x \in F} g(|p-x|)\mathrm{conv}(x), \end{aligned}$$

where x is the individual filter samples in the screen-space support F of the filter (e.g., a 9×9 pixel region), $K(p)$ is the normalization factor, and g is a spatial-filter kernel (e.g., a Gaussian). As a pixel becomes more converged, we shrink the screen-space filter support smoothly, using the shrinking factor s:

$$s(p) = \frac{\max(c_{\mathrm{adaptive}} - \mathrm{conv}(p), 0)}{c_{\mathrm{adaptive}}},$$

Figure 1.6. Rotating dragon, closeups of the marked region are shown. TSSAO without filter (middle) and with our filter (right). Note that the filter is applied only in the noisy regions, while the rest stays crisp.

so that when convergence has reached c_{adaptive}, we turn off spatial filtering completely. We found the setting of c_{adaptive} to be perceptually uncritical (e.g., a value of 0.2 leads to unnoticeable transitions).

The influence of the adaptive convergence-aware filter on the quality of the TSSAO solution is shown in Figure 1.6.

Adaptive sampling. Though spatial filtering can reduce noise, it is more effective to provide additional input samples in undersampled regions. Or, to put it differently, once the AO has reached sufficient convergence, we can just reuse the computed solution, thus using fewer samples in regions that are not undersampled. We adapt the number k of new AO samples per frame as a function of convergence. Note that these AO samples are completely unrelated to the screen-space samples used for spatial filtering in the previous section, where the kernel size is adapted instead of changing the number of samples.

It is necessary to generate at least a minimum number of samples for the same reasons that we clamp $w_f(p)$ in Equation (1.6) (i.e., to avoid blurring artifacts introduced by bilinear filtering). Furthermore, a certain number of samples is required for detecting invalid pixels due to changing neighborhoods (Section 1.4.2). In order to introduce a minimum amount of branching, we chose a simple two-

Parameter name	**Value**
Initial samples k_1	32
Converged samples k_2	8–16
Threshold c_{adaptive}	0.2
Threshold c_{spatial}	0.3
Threshold c_{rot}	0.5
Smooth invalidation factor S	15–30
Maximum weight $w_{\max}$	500–1500
Filter width F	5x5

Table 1.1. Recommended parameters for the TSSAO algorithm.

stage scheme, with k_1 samples if $\text{conv}(p) < c_{\text{spatial}}$ and k_2 samples otherwise (refer to Table 1.1 for a list of parameters actually used in our implementation).

This requires a variable number of iterations of the AO loop in the shader. Since disoccluded regions are often spatially coherent (as can be seen in Figure 1.4(b)), the dynamic branching operations in the shader are quite efficient on today's graphics hardware.

1.4.4 Optimizations

These optimizations of the core algorithm allow for faster frame rates and better image quality due to greater precision.

Random noise. As in most AO approaches, we rotate the sampling pattern by a different random vector for each input pixel. This rotation trades banding artifacts due to undersampling for noise. However, this leads to a surprisingly large performance hit, supposedly due to texture-cache thrashing [Smedberg and Wright 09]. Therefore we turn off the rotation once convergence has reached a certain threshold c_{rot}. Note that this optimization works well in most cases, but can sometimes cause problems for the neighborhood invalidation, resulting in some noticeable artifacts.

Local space. In order to avoid precision errors in large scenes, we store our position values in a local space that is centered at the current view point. These values can be easily transformed into world-space values by passing the previous and the current view point as parameters to the shader.

1.5 SSAO Implementation

We implemented and tested our algorithm using the methods of Fox and Compton [Fox and Compton 08] and Ritschel et al. [Ritschel et al. 09], but it is also possible to implement one of the many alternatives [Bavoil et al. 08, Mittring 07]. In this section we outline the first two algorithms and give some implementation hints based on our experience.

Algorithm of Ritschel et al. The SSAO method of Ritschel et al. uses a 3D sampling kernel, and a depth test to query the sample visibility, thus implementing the contribution function in Equation (1.3). This is in contrast to the original Crytek implementation [Mittring 07], which does not use the incident angle to weight the sample. In order to get a linear SSAO falloff, we use a sample distribution that is linear in the sampling-sphere radius. Note that we use a constant falloff function $D(x) = 1$, in this case—the falloff is caused only by the distribution of the sample points. The differences to the ray-traced AO are mainly caused by the screen-space discretization of the scene.

Algorithm of Fox and Compton. The algorithm of Fox and Compton samples the depth buffer around the pixel to be shaded and interprets these samples as small patches, similar to radiosity. While not physically accurate, it often gives a pleasing visual result because it preserves more small details when using large kernels for capturing low-frequency AO. On the downside, this method is prone to reveal the underlying tessellation. As a remedy, we do not count samples at grazing angles of the hemisphere (i.e., where the cosine is smaller than a given ϵ). We used Equation (1.11) to implement the algorithm:

$$C(p, s_i) = \frac{\max(\cos(s_i - p, n_p), 0)}{\max(\epsilon, |s_i - p|)}. \tag{1.11}$$

The main difference from other SSAO methods is that each sample is constructed on a visible surface, and interpreted as a patch, whereas in Equation (1.11)), samples are used to evaluate the visibility function. The denominator represents a linear falloff function $D(.)$, where we also guard against zero sample distance.

The screen-space sampling radius is defined by projecting a user-specified world-space sampling radius onto the screen, so that samples always cover roughly similar regions in world space. When the user adjusts the world-space radius, the intensity of each sample needs to be scaled accordingly in order to maintain a consistent brightness.

Sample generation. In order to obtain the samples used for SSAO, we first compute a number of random samples ζ in the range $[0 \ldots 1]$ using a Halton sequence. For the method of Fox and Compton, we use 2D (screen-space) samples uniformly distributed on a disc of user-specified size. These samples are then projected from screen space into world space by intersecting the corresponding viewing rays with the depth buffer and computing their world space position. We generate the screen-space samples s_i from 2D Halton samples $\zeta_{x,y}$ using Equation (1.12):

$$\begin{aligned} \alpha &= 2\pi\zeta_x, \\ r &= \sqrt{\zeta_y}, \\ s_i &= (r\cos(\alpha), r\sin(\alpha)). \end{aligned} \tag{1.12}$$

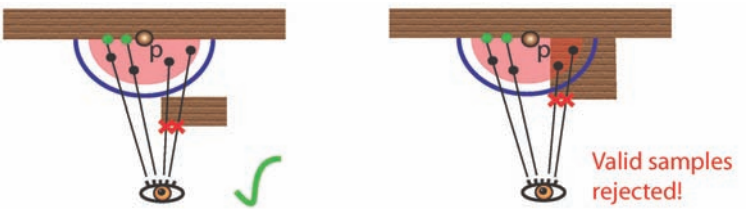
For the method of Ritschel et al., we use 3D hemispherical samples generated in the tangent space of a surface point. From 3D Halton samples $\zeta_{x,y,z}$, hemispherical samples s_i are created using

$$\begin{aligned} \alpha &= 2\pi\zeta_x, \\ r &= \zeta_y, \\ s_i &= (r\cos(\alpha)\sqrt{1-\zeta_z}, r\sin(\alpha)\sqrt{1-\zeta_z}, r\sqrt{\zeta_z}). \end{aligned} \tag{1.13}$$

This formula uses the variables ζ_x and ζ_z to compute a point on the unit sphere (i.e., a random direction), and ζ_y to set the sample distance r. Note that in order to use variance-reducing *importance sampling* (rather than uniform sampling), this formula creates a distribution proportional to the cosine-weighted solid angle. Furthermore, Szirmay-Kalos et al. [Szirmay-Kalos et al. 10] have shown that a uniform distribution of samples along the distance r corresponds to a linear falloff function $D(.)$ (refer to Equation (1.1)) of the occluder influence with respect to r. Using this importance sampling scheme, we simply have to count the numbers of samples that pass the depth test during SSAO shading.

Maximum allowed sample distance. If the distance from the pixel center to the intersection point with the depth buffer is too large, a sample is very likely to cause wrong occlusion (refer to Figure 1.7(a)). However, introducing a *maximum allowed sample radius* and setting it to a too small value can cause samples where valid occlusion is missed (refer to Figure 1.7(b)). This is because they are projected to a location outside the allowed sample radius. We set the maximum

(a) If the allowed sample distance is unrestricted, shadows are cast from disconnected objects (1). If it is set equal to the sampling radius, some valid samples are not counted, resulting in overbright SSAO (2). Allowing 2 times the radius (3) is a good trade-off and closest to a ray-traced solution (REF).



(b) 2D illustration of the issue arising when setting the maximum allowed sample distance (shown in blue, sampling radius shown in red) too small. While the samples (shown in black) which are projected to the disconnected surface are correctly rejected (left), this configuration also rejects valid samples (right).

Figure 1.7. The effect of the maximum allowed sample distance (from the shaded pixel to the depth buffer intersection).

allowed sample radius to reject those samples where the distance is more than two times larger than the sampling radius of the SSAO kernel. This trade-off largely prevents incorrect shadowing of distant and disconnected surfaces caused by objects in the foreground, while still accounting for correct occlusion in the vicinity of the current pixel.

Frame buffer borders. A problem inherent in SSAO is the handling of samples that fall outside the frame buffer borders (requested by fragments near the border). The simplest solution is to settle for reusing the values at the border by using clamp-to-edge. To avoid artifacts on the edges of the screen due to the missing depth information, we can optionally compute a slightly larger image than we finally display on the screen. It is sufficient to extend about 5–10% on each side of the screen depending on the size of the SSAO kernel and the near plane.

1.6 Results

We implemented the proposed algorithm in OpenGL using the Cg shading language. As test scenes, we used two models of different characteristics (shown in Figure 1.8): (a) the Sibenik cathedral and (b) the Vienna city model. Both scenes were populated with several dynamic objects. The walk-through sequences taken for the performance experiments are shown in the accompanying videos. Note that most SSAO artifacts caused by image noise are more distracting in animated sequences, hence we point interested readers to these videos which can be downloaded from http://www.cg.tuwien.ac.at/~matt/tssao/. For all of our tests we used an Intel Core 2 processor at 2.66 GHZ (using 1 core) and an NVIDIA GeForce 280GTX graphics board. To achieve sufficient accuracy in large-scale scenes like Vienna, we use 32-bit depth precision. Both the ambient occlusion buffer and the SSAO texture are 32-bit RGBA render targets.

Generally TSSAO provides finer details and fewer noise artifacts. This can be seen in Figure 1.1 for a static scene (using the method of Fox and Compton),

(a) (b) (c)

Figure 1.8. Used test scenes: (a) Sibenik cathedral (7,013,932 vertices) and (b) Vienna (21,934,980 vertices) in the streets and (c) from above.

Figure 1.9. Close-up of a distant dragon in Sibenik cathedral at 1600×1200 resolution: SSAO using 32 samples at 12 FPS (left close-up); TSSAO using 8–32 samples per frame at 26 FPS (right close-up).

where we compare TSSAO to SSAO with a weak and a strong blur filter, which gives a high or low weight, respectively, to discontinuities. Furthermore, we compare TSSAO to a reference solution using 480 samples per frame, which was the highest number of samples our shader could compute in a single frame. Notice that the TSSAO method is visually very close to the reference solution, to which it converges after a short time.

Figure 1.9 shows that the method also works for high-resolution images. The TSSAO algorithm provides good quality even for fine details in the background. Figure 1.10 shows a capture of a deforming cloak of an animated character. Although deforming objects are difficult to handle with temporal coherence, it can be seen that TSSAO significantly reduces the surface noise. We used the method of Fox and Compton for Figure 1.9, and the method of Ritschel et al. for Figure 1.10.

In terms of visual image-quality, TSSAO performs better than SSAO in all our tests. It corresponds to at least a 32-sample SSAO solution (since 32 samples are always used for disocclusions), while the converged state takes up to several hundred samples into account. Note that a similar quality SSAO solution would

Figure 1.10. Close-up of a deforming cloak: SSAO using 32 samples (middle) and TSSAO using 8–32 samples (right). Notice that the surface noise (causing severe flickering artifacts when animated) is reduced with TSSAO.

	Vienna (fps)			Sibenik cathedral (fps)		
resolution	**SSAO**	**TSSAO**	**Deferred**	**SSAO**	**TSSAO**	**Deferred**
1600x1200	14	29	73	10	12	38
1024x768	30	51	97	21	25	65
800x600	42	63	102	29	34	67
800x600 h	65	77		47	49	

Table 1.2. Average timings for the two walk-through sequences shown in the videos for 32-bit precision render targets using full and half resolution. We compare standard SSAO, our method (TSSAO), and deferred shading without SSAO as a baseline. For SSAO we used 32 samples in all scenes. For TSSAO we used 8 (16)–32 samples in Vienna (Sibenik cathedral).

be prohibitively slow. In terms of correctness, however, we have to keep in mind that using a smooth invalidation causes the algorithm to deviate from a correct solution for the benefit of a better visual impression.

Timings. Table 1.2 shows average timings of our walk-throughs, comparing our method (TSSAO) with SSAO without temporal coherence and the performance-baseline method, deferred shading without SSAO. TSSAO uses 8 06 16 samples, respectively, when converged and 32 otherwise for Vienna and for Sibenik cathedral, whereas SSAO always uses 32 samples. In our tests TSSAO was always faster than SSAO, for full and for half-resolution SSAO computation. Note that, after convergence has been reached, TSSAO neither applies spatial filtering nor the random rotations of the sampling-filter kernel (refer to Section 1.4.4).

Figure 1.11 shows the frame time variations for both walk-throughs. Note that online occlusion culling [Mattausch et al. 08] is enabled for the large-scale Vienna model, and thus the frame rate for the baseline deferred shading is quite high for such a complex model. The frame-rate variations for TSSAO stem from the fact that the method generates adaptively more samples for recently dis-

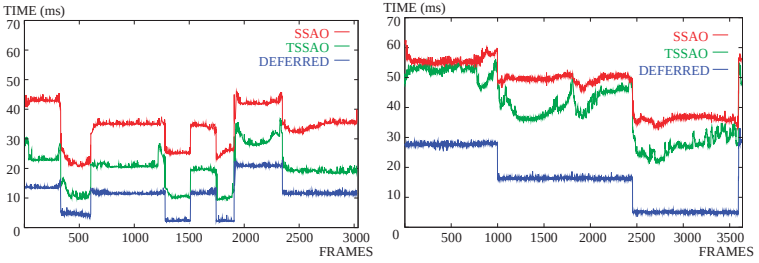


Figure 1.11. Frame times of the Vienna (left) and the Sibenik cathedral walk-through (right) at resolution 1024×768, using 32-bit precision render targets.

occluded regions. The frame times of TSSAO are similar to SSAO for frames where dynamic objects are large in screen space. For more static parts of the walk-throughs, TSSAO is significantly faster.

1.7 Discussion and Limitations

In the case of deforming objects, most of the visual improvements compared to conventional SSAO come from the smooth invalidation (refer to the close-up of the cloak in Figure 1.10). Carefully adjusting the smooth invalidation factor is quite important here, and using this optimization too loosely can result in artifacts such as a noticeable dark trail following moving objects.

There is definitely a limit to exploiting temporal coherence once the objects are moving or deforming too quickly. In such a case, our algorithm will deteriorate to the quality of conventional SSAO. Also, invalidation may fail in cases of thin (with respect to the SSAO kernel size), quickly moving structures, which are missed by the invalidation algorithm. Likewise, a very large kernel size can also cause problems with the invalidation, because the sampling could just miss events that cause changes to the AO.

The main targets of this algorithm are real-time visualization and games. The adaptive sampling optimization cannot guarantee constantly better frame times than conventional SSAO, and disturbing fluctuations in the frame time can happen. However, we have to keep in mind that games usually undergo extensive play-testing in order to avoid annoying frame rate drops. Faster SSAO in most frames is useful, because more time for other effects is available.

Also, note that adaptive sampling, which is responsible for frame rate variations, is a feature of the algorithm that can be disabled, thus falling back to the speed of conventional SSAO, but still obtaining significantly improved image quality. Furthermore, the major purpose of this algorithm is to speed up the standard case of moderate dynamic movements.

1.8 Conclusions

We have presented a screen-space ambient-occlusion algorithm that utilizes reprojection and temporal coherence to produce high-quality ambient occlusion for dynamic scenes. Our algorithm reuses available sample information from previous frames if available, while adaptively generating more samples and applying spatial filtering only in the regions where insufficient samples have been accumulated. We have shown an efficient new pixel validity test for shading algorithms that access only the affected pixel neighborhood. Using our method, such shading methods can benefit from temporal reprojection also in dynamic scenes with animated objects.

Bibliography

[Bavoil et al. 08] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. "Image-Space Horizon-Based Ambient Occlusion." In *ACM SIGGRAPH 2008 Talks*, *SIGGRAPH '08*, pp. 22:1–22:1. New York: ACM, 2008.

[Cook and Torrance 82] Robert L. Cook and Kenneth E. Torrance. "A Reflectance Model for Computer Graphics." *ACM Trans. Graph.* 1 (1982), 7–24.

[Eisemann and Durand 04] Elmar Eisemann and Frédo Durand. "Flash Photography Enhancement via Intrinsic Relighting." *ACM Trans. Graph.* 23 (2004), 673–678.

[Fox and Compton 08] Megan Fox and Stuart Compton. "Ambient Occlusive Crease Shading." *Game Developer Magazine*, 2008.

[Landis 02] Hayden Landis. "Production-Ready Global Illumination." In *Siggraph Course Notes, Vol. 16*. New York: ACM, 2002.

[Mattausch et al. 08] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. "CHC++: Coherent Hierarchical Culling Revisited." *Computer Graphics Forum (Proceedings of Eurographics 2008)* 27:2 (2008), 221–230.

[Mittring 07] M. Mittring. "Finding Next Gen - CryEngine 2." In *ACM SIGGRAPH 2007 courses*, *SIGGRAPH '07*, pp. 97–121. New York: ACM, 2007.

[Nehab et al. 07] Diego Nehab, Pedro V. Sander, Jason Lawrence, Natalya Tatarchuk, and John R. Isidoro. "Accelerating real-time shading with reverse reprojection caching." In *Proceedings of the Eurographics Symposium on Graphics Hardware 2007*, pp. 25–35. Aire-la-Ville, Switzerland: Eurographics Association, 2007.

[Ritschel et al. 09] Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. "Approximating Dynamic Global Illumination in Image Space." In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, pp. 75–82. New York: ACM, 2009.

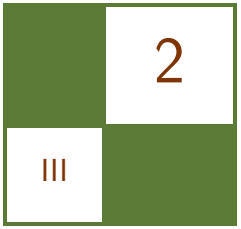
[Rosado 07] Gilberto Rosado. "Motion Blur as a Post-Processing Effect." In *GPU Gems 3*, pp. 575–576. Reading, MA: Addison Wesley, 2007.

[Scherzer et al. 07] Daniel Scherzer, Stefan Jeschke, and Michael Wimmer. "Pixel-Correct Shadow Maps with Temporal Reprojection and Shadow Test Confidence." In *Proceedings of the Eurographics Symposium on Rendering 2007*, pp. 45–50. Aire-la-ville, Switzerland: Eurographics Association, 2007.

[Smedberg and Wright 09] Niklas Smedberg and Daniel Wright. "Rendering Techniques in Gears of War 2." In *Proceedings of the Game Developers Conference '09*, 2009.

[Szirmay-Kalos et al. 10] Laszlo Szirmay-Kalos, Tamas Umenhoffer, Balazs Toth, Laszlo Szecsi, and Mateu Sbert. "Volumetric Ambient Occlusion for Real-Time Rendering and Games." *IEEE Computer Graphics and Applications* 30 (2010), 70–79.

[Wang and Hickernell 00] Xiaoqun Wang and Fred J. Hickernell. "Randomized Halton Sequences." *Mathematical and Computer Modelling* 32 (2000), 887–899.



Level-of-Detail and Streaming Optimized Irradiance Normal Mapping

Ralf Habel, Anders Nilsson, and Michael Wimmer

2.1 Introduction

Light mapping and normal mapping are the most successful shading techniques used in commercial games and applications today, because they require few resources and result in a significant increase in the quality of the rendered image. While light mapping stores global, low-frequency illumination at sparsely sampled points in a scene, normal maps provide local, high-frequency shading variation at a far higher resolution (see Figure 2.1).

The problem with combining the two methods is that light maps store irradiance information for only one normal direction—the geometric surface normal—and therefore cannot be evaluated using the normals stored in a normal map. To overcome this problem, the irradiance (i.e., the incoming radiance integrated over the hemisphere) has to be precalculated for all possible normal map directions at every sample point. At runtime, this (sparse) *directional irradiance* signal can be reconstructed at the (dense) sampling positions of the normal map through interpolation. The final irradiance is calculated by evaluating the interpolated directional irradiance using the normal vector from the normal map.

Because such a (hemispherical) directional irradiance signal is low-frequency in its directionality, it can be well represented by smooth lower-order basis functions. Several bases have been successfully used in games such as spherical harmonics in *Halo 3* [Chen and Liu 08], or the hemispherical *Half-Life 2* basis [McTaggart 04]. It has been shown [Habel and Wimmer 10] that the hemispherical $\mathcal{H}$-basis provides the overall best representation compared to all other bases

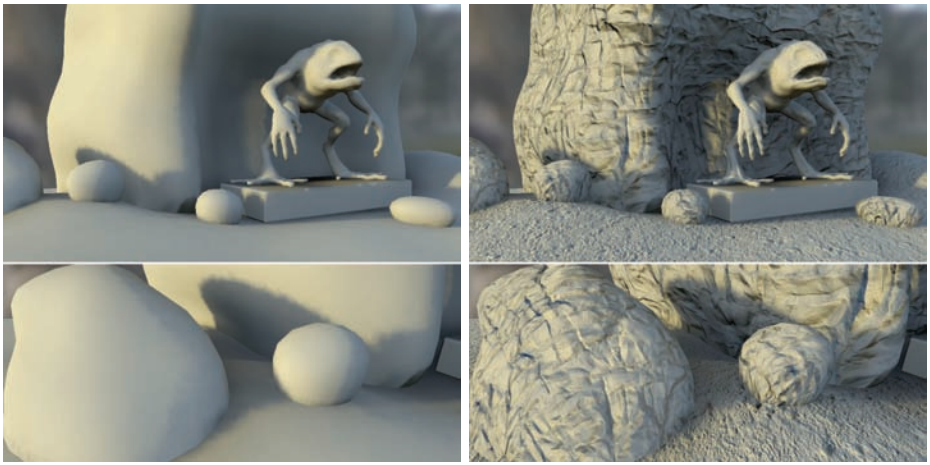


Figure 2.1. A scene without albedo maps showing the difference between light mapping (left) and irradiance normal mapping (right).

because it is directly constructed to carry hemispherical directional irradiance signals and incorporates the same advantages that spherical harmonics have.

However, none of the options to represent the directional irradiance signal provides an efficient and resource-optimal way to define increments of detail for shader level-of-detail (LOD) or for streaming of textures. To eliminate those drawbacks, we introduce a modification to the $\mathcal{H}$-basis that maintains its high accuracy and fast evaluation. In contrast to any other known basis representation, the modified $\mathcal{H}$-basis explicitly contains the traditional light map as one of its coefficients and is therefore an extension of light mapping rather than a replacement, providing the light map as the lowest shader LOD.

To apply the optimized irradiance normal mapping, we first derive all necessary equations and calculations, after which we discuss implementation issues such as correct tangent space, color spaces, or compression.

2.2 Calculating Directional Irradiance

The directional irradiance of a surface point $\vec{x}$ is defined as

$$E(\vec{x}, \vec{n}) = \int_{\Omega^+} L(\vec{x}, \vec{\omega}) \max(0, \vec{n} \cdot \vec{\omega}) d\vec{\omega} \tag{2.1}$$

and covers all possible surface normals $\vec{n}$ in the upper hemisphere Ω_+, with L being the incoming radiance. Because we do not want to keep track of the orientation of the hemispheres at every point, the hemispheres are defined in

tangent space, (i.e., around the interpolated surface normal and (bi)tangent), which is also the space where the normal maps are defined.

To determine the incoming directional irradiance, we need to calculate the radiance $L(\vec{x}, \omega)$ in a precomputation step similar to traditional light mapping. Any method that creates a radiance estimate such as shadow mapping, standard ray tracing, photon mapping [Jensen 96], final gathering, or path tracing [Kajiya 86] can be used, and existing renderers or baking software can be applied.

Given the radiance $L(\vec{x}, \omega)$, calculating $E(\vec{x}, \vec{n})$ (Equation (2.1)) for a point $\vec{x}$ corresponds to filtering L with a diffuse (cut cosine) kernel. Doing this in Euclidean or spherical coordinates is prohibitively expensive, because we have to filter a large number of surface points. Instead, we use spherical harmonics as an intermediate basis in which the filtering can be done much more efficiently. Spherical harmonics are orthonormal basis functions that can approximate any spherical function. A comprehensive discussion of spherical harmonics can be found in [Green 03] and [Sloan 08]. Unfortunately, different definitions exist; to avoid confusion, we use the definition without the Condon-Shortley phase, as shown in Appendix A for our calculations.

As shown by [Ramamoorthi and Hanrahan 01], a spherical directional irradiance signal is faithfully represented with three spherical harmonics bands (nine coefficients per color channel). Therefore, we need only to use spherical harmonics up to the quadratic band.

First, we rotate the sampled radiance of a surface point into the tangent space and expand it into spherical harmonics coefficients s_m^l by integrating against the spherical harmonics basis functions Y_m^l over the upper hemisphere Ω_+:

$$s_m^l = \int_{\Omega_+} L(\vec{\omega}) Y_m^l(\vec{\omega}) d\vec{\omega}.$$

In almost all cases, the coefficients are calculated using Monte Carlo Integration [Szirmay-Kalos]:

$$s_m^l \approx \frac{2\pi}{N} \sum_{i=1}^{N} L(\vec{\omega_i}) Y_m^l(\vec{\omega_i}),$$

where N is the number of hemispherical, equally distributed, radiance samples $L(\vec{\omega_i})$. More advanced methods, such as importance sampling, can be applied, as long as a radiance estimate represented in spherical harmonics is calculated. The diffuse convolution reduces to an almost trivial step in this representation. Following [Ramamoorthi and Hanrahan 01], applying the Funk-Hecke-Theorem, integrating with a cut cosine kernel corresponds to multiplying the coefficients of each band with a corresponding factor a^l:

$$a^0 = 1 \qquad a^1 = \frac{2}{3} \qquad a^2 = \frac{1}{4},$$

to arrive at the final directional irradiance signal represented in spherical harmonics:

$$E_{\text{SH}}(\vec{n}) = \sum_{l,m} s_m^l Y_m^l(\vec{n}). \tag{2.2}$$

We have built the necessary division by π for the exitant radiance into the diffuse kernel so we do not have to perform a division at runtime.

By storing nine coefficients (respectively, 27 in the trichromatic case) at each surface point, in either textures or vertex colors, we can calculate the final result at runtime by looking up the normal from the normal map and we can calculate the final irradiance by evaluating Equation (2.2). However, we are not making the most efficient use of the coefficients since the functions are evaluated only on the upper hemisphere Ω_+, and not the full sphere. The created directional irradiance signals can be better represented in a hemispherical basis.

2.3 $\mathcal{H}$-Basis

The $\mathcal{H}$-basis was introduced in [Habel and Wimmer 10] and forms an orthonormal hemispherical basis. Compared to all other orthonormal hemispherical bases, such as [Gautron et al. 04] or [Koenderink et al. 96], the $\mathcal{H}$-basis consist of only polynomial basis functions up to a quadratic degree and therefore shares many properties with spherical harmonics. Some of the basis functions are actually the same basis functions as those used in spherical harmonics, but re-normalized on the hemisphere, which is why the $\mathcal{H}$-basis can be seen as the counterpart of spherical harmonics on the hemisphere up to the quadratic band.

The basis is explicitly constructed to carry hemispherical directional irradiance signals and can provide a similar accuracy with only six basis functions compared to nine needed by spherical harmonics, and a higher accuracy than any other hemispherical basis [Habel and Wimmer 10]. These basis functions are:

$$\begin{aligned}
H^1 &= \frac{1}{\sqrt{2\pi}}, \\
H^2 &= \sqrt{\frac{3}{2\pi}} \sin\phi \sin\theta = \sqrt{\frac{3}{2\pi}} y, \\
H^3 &= \sqrt{\frac{3}{2\pi}} (2\cos\theta - 1) = \sqrt{\frac{3}{2\pi}} (2z - 1), \\
H^4 &= \sqrt{\frac{3}{2\pi}} \cos\phi \sin\theta = \sqrt{\frac{3}{2\pi}} x,
\end{aligned}$$

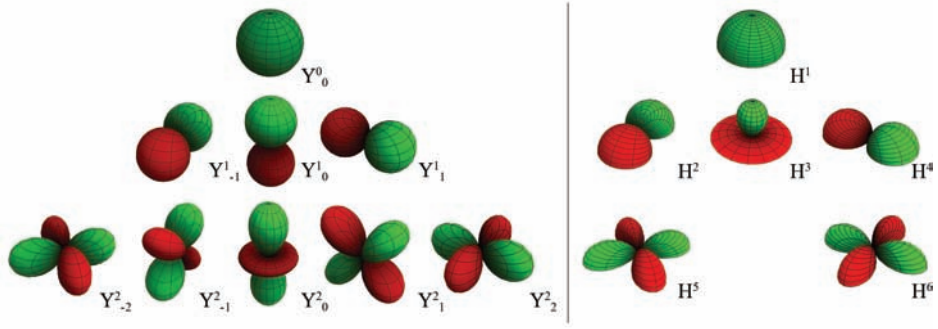


Figure 2.2. Spherical Harmonics basis functions (left) compared to the $\mathcal{H}$-basis functions (right). Green are positive and red are negative values.

$$H^5 = \frac{1}{2}\sqrt{\frac{15}{2\pi}}\sin 2\phi \sin^2\theta = \sqrt{\frac{15}{2\pi}}xy,$$
$$H^6 = \frac{1}{2}\sqrt{\frac{15}{2\pi}}\cos 2\phi \sin^2\theta = \frac{1}{2}\sqrt{\frac{15}{2\pi}}(x^2 - y^2).$$

Please note that compared to [Habel and Wimmer 10], the negative signs caused by the Condon-Shortley phase have been removed in the basis functions H^2 and H^4 for simplicity and for consistency with the spherical harmonics definitions. A visual comparison of the spherical harmonics basis functions to the $\mathcal{H}$-basis can be seen in Figure 2.2.

2.3.1 Modified $\mathcal{H}$-Basis

The band structure of the $\mathcal{H}$-basis (similar to spherical harmonics) provides a natural way for controlling the level of approximation of the directional irradiance signal. For example, given a normal from the normal map, we can use only the first four coefficients (corresponding to the constant and all linear basis functions) to calculate the irradiance. If a higher accuracy is demanded, we can simply use the two quadratic functions H^5 and H^6 in addition to the other four basis functions.

Another level of detail that is very important, especially for objects that are far away, is to not use a normal map at all, but to use a standard light map in order to avoid the loading or streaming of both the used normal map as well as coefficients for the $\mathcal{H}$-basis. A light map corresponds to irradiance values calculated for the geometric normal, which is defined in tangent space as $\vec{n}_g = (0, 0, 1)$ ($\theta = 0$ in spherical coordinates), so the light map value is actually $E(\vec{n}_g)$. If E is represented in the $\mathcal{H}$-basis, we can see that only H^1 and H^3 evaluate to a nonzero value $\vec{n}_g$, so only the corresponding two coefficients

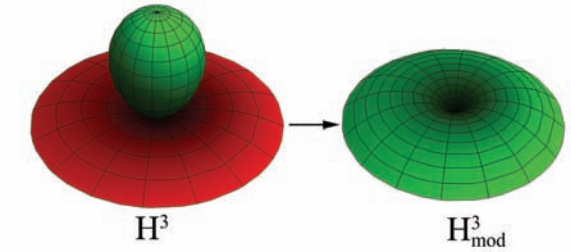


Figure 2.3. The original basis function H^3 (left), and its replacement H^3_{mod} (right). In contrast to H^3, H^3_{mod} is always 0 for $\vec{n}_g = (0, 0, 1)$.

are required. But this is still less efficient than standard light mapping, which requires only one coefficient (i.e., texture). Thus, using the $\mathcal{H}$-basis, we have to load twice the amount of data in order to arrive at the same result we would achieve with light mapping. The question arises, can we combine both light mapping and irradiance normal mapping in one framework?

First, we note that while the $\mathcal{H}$-basis is an orthonormal basis, orthonormality is not required for representing directional irradiance. Thus we can sacrifice this property for the sake of another advantage. As we have seen, all but two of the basis functions always evaluate to 0 when evaluated in the direction of $\vec{n}_g$, but the optimal case would be to have only one contributing function in this case. We can achieve this goal by redefining the basis function H^3 as

$$H^3_{\text{mod}} = \sqrt{\frac{3}{2\pi}}(1 - \cos\theta) = \sqrt{\frac{3}{2\pi}}(1 - z),$$

which is a function that is still linearly independent of the other basis functions, and depends only on z ($\cos\theta$ in spherical coordinates) like H^3. However, this function also evaluates to 0 in the direction of the geometric normal $\vec{n}_g$ (see Figure 2.3). Through this modification, the constant basis function H^1 now represents the traditional light map, while we still maintain the high accuracy of the original $\mathcal{H}$-basis when using all coefficients. Therefore we have constructed a set of basis functions that extends the standard light map approach rather than replacing it.

2.3.2 Expansion into the Modified $\mathcal{H}$-Basis

In Section 2.2, we precomputed the directional irradiance in the spherical harmonics basis as E_{SH} (Equation (2.2)). Transforming E_{SH} into the more efficient, modified $\mathcal{H}$-basis requires another precomputation step. We calculate a transformation matrix by projecting the spherical harmonics basis functions into the

modified $\mathcal{H}$-basis [Sloan 08], resulting in

$$T_{\mathcal{H}_{\text{mod}}} = \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & \sqrt{\frac{3}{2}} & 0 & 0 & 0 & \frac{3}{4}\sqrt{\frac{5}{2}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & \frac{3\sqrt{\frac{5}{2}}}{8} & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & -\frac{1}{2}\sqrt{\frac{15}{2}} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & \frac{3\sqrt{\frac{5}{2}}}{8} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \tag{2.3}$$

We arrive at the coefficient vector for the modified $\mathcal{H}$-basis $\vec{h} = (h_1, h_2, .., h_6)$ by multiplying the serialized spherical harmonics coefficient vector $\vec{s} = (s_0^0, s_{-1}^1, s_0^1, .., s_2^2)$ with the matrix $T_{\mathcal{H}_{\text{mod}}}$. This matrix multiplication also automatically extracts the light map into the coefficient for the constant basis function h_1.

2.3.3 Runtime Evaluation

Up to now, we have precomputed the directional irradiance at different surface points and represented it in the modified $\mathcal{H}$-basis defined in the tangent space of the surface points. The necessary coefficients are transported either on a per-vertex basis or using coefficient texture maps. In the trichromatic case, this results in one color per vertex or one texture per coefficient. As the normal map is also defined in tangent space, we simply need to evaluate the basis functions in the direction of the normal map normal ($\vec{n}$) and weight them by the interpolated coefficients

$$E_{\mathcal{H}_{\text{mod}}}(\vec{n}) = \sum_{i=1}^{n} h_i H_{\text{mod}}^i(\vec{n})$$

for every surface point $\vec{x}$. Please note that $H_{\text{mod}}^i(\vec{n}) = H^i(\vec{n})$ except for the modified basis function H_{mod}^3. Similar to light mapping, the result can be modulated by a lookup from an albedo texture and combined with additional shading effects, such as specular lighting.

2.4 Implementation

In the previous sections, we derived all necessary expressions and equations to precompute the data and evaluate it at runtime. In the following sections, we will discuss practical issues that occur in an application of irradiance normal mapping, such as proper generation, representation, and compression of the data. In general, more sophisticated methods can be applied to compress and distribute the coefficient maps [Chen and Liu 08, Sloan et al. 03], but this usually requires the use of proprietary formats that are beyond this discussion.

2.4.1 Precomputing Directional Irradiance

Solving the radiance distribution in a full scene is a complex task in itself and is usually done using commercial renderers or game-engine-native baking software. In our case, we use the Maya-based [Autodesk 10] rendering and baking plug-in, Turtle 5.1 [Illuminate Labs 10]. Turtle can be customized to bake into any basis through LUA scripting. A script implementing the discussed methods is included in the demo and can also be executed with the trial version of Turtle 5.1.

Because we construct a physically correct representation of the directional irradiance, we need to take into account that all calculations are performed in linear color space. Therefore, irradiance normal mapping lends itself to be used in a linear rendering pipeline using sRGB texture lookups and frame buffers, or gamma-correct high dynamic range (HDR) tone mapping where necessary. If a non-gamma-correct pipeline is used, the result simply needs to be exponentiated with $1/2.2$.

Concerning resolutions of coefficient textures, the exact same principles used in standard light mapping apply, and the chosen resolution directly reflects the spatial resolution of the lighting that can be represented.

2.4.2 Directional Irradiance Tangent Space

In almost all practical cases that use texture maps, a second set of texture coordinates that create an unambiguous mapping of the surfaces is used to define the light map, or in our case, coefficient textures. If the unambiguous texture coordinates are used only for a light map, the orientation of the corresponding tangent spaces spanned by the different texture coordinate sets is irrelevant for the light map evaluation. This changes with irradiance normal mapping because it may happen that both tangent spaces have different orientations, and the normal map resides in a different tangent space than the directional irradiance.

Fortunately, both texture coordinate sets share the same vertices and geometric normals, as well as interpolation in the rendering pipeline. By simply using the tangent and bitangent of the normal map tangent space in the directional irradiance texture coordinates during the precomputation, a tangent space for the directional irradiance texture coordinates is constructed that is correctly oriented with the normal- and albedo-map tangent space.

Effectively, the directional irradiance texture coordinate set is used only to define the surface points where it is calculated and stored, while the normal map tangent space defines the tangent and bitangent for both spaces, resulting in a correctly aligned irradiance signal relative to the normal map. Turtle provides the option to use the (bi)tangent of another texture coordinate set in its baking menu.

Since both tangent spaces are aligned, when evaluating the directional irradiance at runtime, we need neither the normals nor the (bi)tangents of any tangent space but solely the texture coordinate sets to define the texture lookups.

2.4.3 Texture Normalization and Range Compression

The output of the precomputations are floating point values which, similar to spherical harmonics, can also be negative. Though we could use floating point texture formats to represent the sign and full dynamic range of the directional irradiance, this option is prohibitively expensive in most cases, though it leads to the best possible output.

To transport the coefficient textures efficiently, we treat the map containing h_1, which is also the light map, differently, because it can never get negative, in contrast to the other coefficients. Compared to standard light mapping, we cannot allow h_1 and the other coefficient maps to oversaturate, since this would lead to color-shifting artifacts. To avoid oversaturation, we can simply choose the lighting so that h_1 stays within the range [0..1], or clamp the calculated radiance to this range before the spherical harmonics expansion in order to transport h_1 with an 8-bit texture.

Additionally, because h_1 is a light map, we change the color space from linear to sRGB by exponentiation with $1/2.2$ to avoid quantization artifacts associated with an 8-bit linear representation. As with standard texture maps, we have to interpret the texture as sRGB data when evaluating it at runtime.

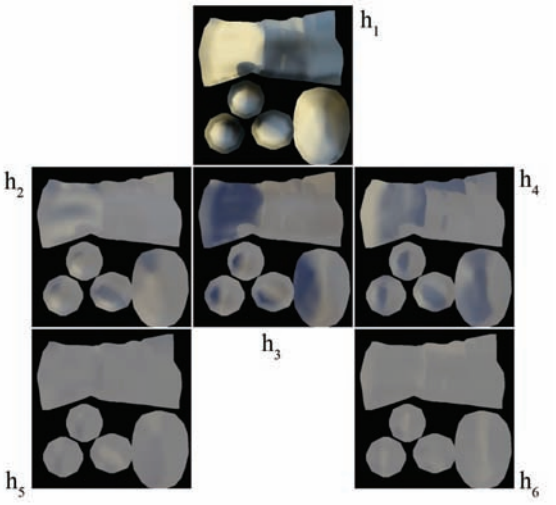


Figure 2.4. A set of coefficient textures. The coefficient h_1 is the standard light map.

```
for each surface point do {

  color SHc[9] //3 bands Spherical Harmonics coefficients

  //Monte-Carlo-Integration in tangent space over
  //the hemisphere to project into SH[] basis functions
  for each radiance sample L(direction) in N do {
    for each SHc do {
      SHc[]  +=  L(direction)*SH[](direction)
    }
  }
  SHc[] = (2*PI/N)*SHc[]

  //Diffuse convolution
  SHc[0] = SHc[0]
  SHc[1,2,3] = 2.0/3.0*SHc[2,3,4]
  SHc[4,5,6,7,8] = 1.0/4.0*SHc[4,5,6,7,8]

  //Projection into modified H-basis
  color modHc[6] // modified H-basis coefficients

  //Transform matrix
  for each color in modHc[] do {
    modHc[0] =  0.70711*SHc[0]+1.2247*SHc[2]+1.1859*SHc[6]
    modHc[1] =  0.70711*SHc[1]+0.59293*SHc[5]
    modHc[2] = -0.70711*SHc[2]-1.3693*SHc[6]
    modHc[3] =  0.70711*SHc[3]+0.59293*SHc[7]
    modHc[4] =  0.70711*SHc[4]
    modHc[5] =  0.70711*SHc[8]
  }

  //Convert first coefficient to sRGB
  modHc[0] = pow(modHc[0],1/2.2)

  //Range compress rest with global factor of 0.75
  for each modHc[] except modHc[0] do {
    modHc[] = modHc[]+0.75/(2*0.75)
  }

  write modHc[]
}
```

Listing 2.1. Pseudo-code for calculating the coefficients for the modified $\mathcal{H}$-basis including the texture optimizations.

Because the coefficients $h_{2..6}$ can be negative, we perform a similar range compression as done with normal maps, keeping them in linear color space. For a data-independent range compression, if h_1 is in the range [0..1], a rule of thumb is that the other coefficients do not exceed the range of $[-\frac{3}{4}..\frac{3}{4}]$, resulting in a range compression

$$h'_{2..6} = \frac{h_{2..6} + \frac{3}{4}}{2 \cdot \frac{3}{4}}.$$

The range compression factor of $\frac{3}{4}$ may be changed according to the scene and lighting conditions to further increase the accuracy of the coefficient representation over a range compression of $[-1..1]$, as used in normal mapping. Figure 2.4 shows a set of coefficient textures resulting from the calculations shown. Pseudocode showing all precomputation calculations, including the described optimizations, is given in Listing 2.1.

Optimal range compression. A much better way to represent the coefficients is to perform a range compression on a per-shape, per-texture, or even per-channel basis. Calculating both the minimum and maximum value (with negative minima for $h_{2...6}$) or the maximum absolute range (to replace the previously chosen $\frac{3}{4}$) and normalizing the data can increase the quality of the representation significantly. This allows irradiance normal mapping within both standard and HDR frameworks, using just 8-bit textures. Because directional irradiance is a convolved signal, it does not have the extreme values of radiance signals such as HDR environment maps, so this optimized 8-bit representation may be sufficient. Besides the exposition of the minimum/maximum or maximum absolute-range value to the shader, there is almost no overhead since most of the coefficients need to be transported range compressed anyway.

2.4.4 Texture Compression

In a manner similar to that used for light maps, the coefficient maps can be compressed using DXT texture formats. Since we do not need any alpha information, DXT1 compression can deliver a compression rate of 6 : 1. Because we have six coefficient maps, a set of compressed maps therefore requires the same amount of memory as an uncompressed light map. Since we effectively add up several DXT compressed textures, compression artifacts may also add up to an intolerable level. These artifacts can be counteracted by choosing a higher-coefficient map resolution or by modulating the result with an albedo map to obfuscate the color shifts.

2.4.5 Levels of Detail and Streaming

As discussed in Section 2.3.1, the lowest LOD is formed by evaluating only the constant term $h_1 H^1_{\text{mod}}(\vec{n})$. As a mid-range LOD, only the first four coefficients,

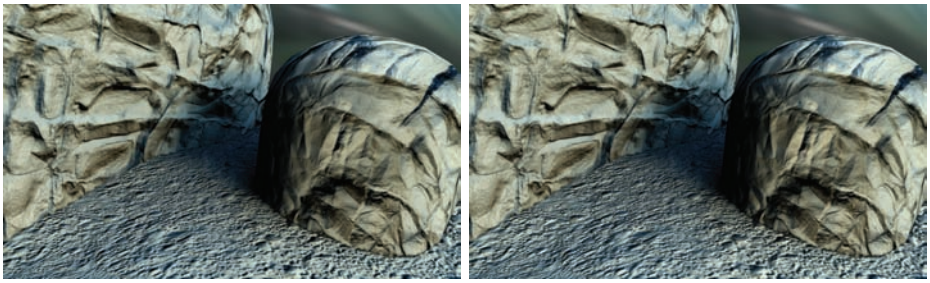


Figure 2.5. Detail of a game scene without albedo texturing using four coefficients (left) and for more accuracy six coefficients (right). The differences can be marginal, though six coefficients show more detail where the normal map normal is perpendicular to the geometric normal.

consisting of the constant and the linear basis functions, can be used. This LOD uses the normal map, so the appearance of a material compared with the lowest LOD may cause popping artifacts. A small distance-based region where the middle LOD is blended with the lowest can suppress those artifacts, as can be seen in the demo.

As highest LOD, a full evaluation using all six coefficient maps can be applied, though the difference between using four or six coefficients can be marginal, and four coefficients may already deliver a perceptually accurate result (see Figure 2.5). Depending on the available resources, the highest LOD can therefore be skipped if the quality of the middle LOD is sufficient. The reason for this behavior is that the quadratic basis functions H^5_{mod} and H^6_{mod} mostly contribute if the normal map normal is perpendicular to the geometric normal.

All levels of details are cumulative, so if textures are streamed, the next higher LOD uses the same textures as the lower one. Also, due to the linearity of the basis functions, the coefficient textures can be used simultaneously even if they are available at different resolutions, switching to an appropriate LOD as soon as some mip-level of a coefficient texture is available. The HLSL code for evaluating the modified $\mathcal{H}$-basis is given in Listing 2.2, including range decompression, gamma-correct color-space lookups, and levels of detail.

```
float3 n = 2*tex2D(normal,texUV)-1; //tangent space normal map

float3 irr =
  0.39894*tex2D(h1,lightUV) //is sRGB lookup ( x^2.2 )
  //stop here for lowest LOD (lightmap)
  +(2*0.75*tex2D(h2,lightUV)-0.75)*0.69099*n.y //not sRGB lookup
  +(2*0.75*tex2D(h3,lightUV)-0.75)*0.69099*(1-n.z)
  +(2*0.75*tex2D(h4,lightUV)-0.75)*0.69099*n.x
```

```
    //stop here for middle LOD
    +(2*0.75*tex2D(h5,lightUV)-0.75)*1.54509*n.x*n.y
    +(2*0.75*tex2D(h6,lightUV)-0.75)*0.77255*(n.x*n.x-n.y*n.y);
    //full evaluation

color = irr*tex2D(albedo,texUV)  //is sRGB lookup ( x^2.2 );

//write color to sRGB frame buffer ( x^(1/2.2) )
```

Listing 2.2. HLSL code for evaluating the modified $\mathcal{H}$-basis, including a modulation with an albedo map. The different levels of detail are created by stopping the irradiance calculation at the shown points.

2.5 Results

We have implemented the described approach in the graphics engine OGRE 1.6.5 [OGRE 10]. The accompanying web materials contain the binaries as well as the full source code of the demo and the Turtle script to bake out coefficient maps with the described optimizations. All levels of detail and texture formats can be directly compared and viewed in both low-dynamic as well as high-dynamic range rendering pipelines with or without albedo textures (see Figure 2.6).

This can act as a reference implementation since any game engine or application that supports light mapping and shaders can be easily modified to support irradiance normal mapping. Besides the precomputation, only several additional textures need to be exposed to the shader compared to a single texture when using light mapping. The shader calculations consist only of a few multiply-adds for range decompression of the textures and to add up the contributions of the basis functions. Both the data as well as the evaluation are lightweight and simple, and are therefore also applicable to devices and platforms that have only a limited set of resources and calculation power.

2.6 Conclusion

We have derived a modification of the $\mathcal{H}$-basis that allows formulating irradiance normal mapping as an extension of light mapping rather than as a replacement by containing the light map of the basis function coefficients. We discussed the efficient calculation and representation of directional irradiance signals in the modified $\mathcal{H}$-basis using spherical harmonics as an intermediate representation for efficient filtering. A description of the accompanying implementation was given, showing the different levels of detail and optimizations for 8-bit textures, such as optimal color spaces and range compression.

Figure 2.6. A full scene without (top) and with (bottom) albedo mapping.

2.7 Appendix A: Spherical Harmonics Basis Functions without Condon-Shortley Phase

$$
\begin{aligned}
Y_0^0 &= \frac{1}{2}\sqrt{\frac{1}{\pi}} \\
Y_{-1}^1 &= \frac{1}{2}\sqrt{\frac{3}{\pi}}\sin\phi\sin\theta = \frac{1}{2}\sqrt{\frac{3}{\pi}}y \\
Y_0^1 &= \frac{1}{2}\sqrt{\frac{3}{\pi}}\cos\theta = \frac{1}{2}\sqrt{\frac{3}{\pi}}z \\
Y_1^1 &= \frac{1}{2}\sqrt{\frac{3}{\pi}}\cos\phi\sin\theta = \frac{1}{2}\sqrt{\frac{3}{\pi}}x \\
Y_{-2}^2 &= \frac{1}{2}\sqrt{\frac{15}{\pi}}\sin 2\phi\sin^2\theta = \frac{1}{2}\sqrt{\frac{15}{\pi}}xy \\
Y_{-1}^2 &= \frac{1}{2}\sqrt{\frac{15}{\pi}}\sin\phi\cos\theta\sin\theta = \frac{1}{2}\sqrt{\frac{15}{\pi}}yz \\
Y_0^2 &= \frac{1}{4}\sqrt{\frac{5}{\pi}}(3\cos^2\theta - 1) = \frac{1}{4}\sqrt{\frac{5}{\pi}}(3z^2 - 1) \\
Y_1^2 &= \frac{1}{2}\sqrt{\frac{15}{\pi}}\cos\phi\cos\theta\sin\theta = \frac{1}{2}\sqrt{\frac{15}{\pi}}zx \\
Y_2^2 &= \frac{1}{4}\sqrt{\frac{15}{\pi}}\cos 2\phi\sin^2\theta = \frac{1}{4}\sqrt{\frac{15}{\pi}}(x^2 - y^2)
\end{aligned}
$$

Bibliography

[Autodesk 10] Autodesk. "Maya. Maya is a registered trademark or trademark of Autodesk, Inc. in the USA and other countries." Available at http://www.autodesk.com, 2010.

[Chen and Liu 08] Hao Chen and Xinguo Liu. "Lighting and Material of Halo 3." In *SIGGRAPH '08: ACM SIGGRAPH 2008 Classes*, pp. 1–22. New York: ACM, 2008.

[Gautron et al. 04] Pascal Gautron, Jaroslav Krivánek, Sumanta N. Pattanaik, and Kadi Bouatouch. "A Novel Hemispherical Basis for Accurate and Efficient Rendering." In *Rendering Techniques*, pp. 321–330. Aire-la-Ville, Switzerland: Eurographics Association, 2004.

[Green 03] Robin Green. "Spherical Harmonic Lighting: The Gritty Details." Available at http://citeseer.ist.psu.edu/contextsummary/2474973/0, 2003.

[Habel and Wimmer 10] Ralf Habel and Michael Wimmer. "Efficient Irradiance Normal Mapping." In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 189–195. New York: ACM, 2010.

[Illuminate Labs 10] Illuminate Labs. "Turtle for Maya." Available at http://www.illuminatelabs.com, 2010.

[Jensen 96] Henrik Wann Jensen. "Global Illumination using Photon Maps." In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pp. 21–30. London: Springer-Verlag, 1996.

[Kajiya 86] James T. Kajiya. "The Rendering Equation." In *SIGGRAPH '86: Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 143–150. New York: ACM, 1986.

[Koenderink et al. 96] Jan J. Koenderink, Andrea J. van Doorn, and Marigo Stavridi. "Bidirectional Reflection Distribution Function Expressed in Terms of Surface Scattering Modes." In *ECCV '96: Proceedings of the 4th European Conference on Computer Vision-Volume II*, pp. 28–39. London, UK: Springer-Verlag, 1996.

[McTaggart 04] G. McTaggart. "Half-Life 2/Valve Source Shading." Technical report, Valve Corporation, 2004.

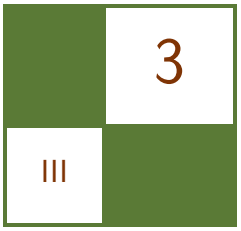
[OGRE 10] OGRE. "OGRE Graphics Engine." Available at http://www.ogre3d.org, 2010.

[Ramamoorthi and Hanrahan 01] Ravi Ramamoorthi and Pat Hanrahan. "An Efficient Representation for Irradiance Environment Maps." In *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 497–500. New York: ACM, 2001.

[Sloan et al. 03] Peter-Pike Sloan, Jesse Hall, John Hart, and John Snyder. "Clustered Principal Components for Precomputed Radiance Transfer." *ACM Trans. Graph.* 22:3 (2003), 382–391.

[Sloan 08] Pete-Pike Sloan. "Stupid Spherical Harmonics (SH) Tricks." Available at http://www.ppsloan.org/publications/StupidSH36.pdf, 2008.

[Szirmay-Kalos] Laszlo Szirmay-Kalos. *Monte Carlo Methods in Global Illumination - Photo-Realistic Rendering with Randomization*. Saarbrücken, Germany: VDM Verlag Dr. Mueller e.K.



Real-Time One-Bounce Indirect Illumination and Shadows using Ray Tracing

Holger Gruen

3.1 Overview

This chapter presents an easily implemented technique for real-time, one-bounce indirect illumination with support for indirect shadows. Determining if dynamic scene elements occlude some indirect light and thus cast indirect shadows is a hard problem to solve. It amounts to being able to answer many point-to-point or region-to-region visibility queries in real time. The method described in this chapter separates the computation of the full one-bounce indirect illumination solution into three phases. The first phase is based on *reflective shadow maps* (RSM) [Dachsbacher and Stamminger 05] and is fully Direct3D 9 compliant. It generates the one-bounce indirect lighting from a kernel of RSM texels without considering blockers of indirect light. The second phase requires Direct3D 11–capable hardware and dynamically creates a three-dimensional grid that contains lists of triangles of the geometry that should act as blockers of indirect light. The third phase traverses the 3D grid built in phase 2, tracing rays to calculate an approximation of the indirect light from RSM texels that are blocked by geometry. Finally, the result of the third phase is subtracted from the result of the first phase to produce the full indirect illumination approximation.

3.2 Introduction

Real-time indirect illumination techniques for fully dynamic scenes are an active research topic. There are a number of publications (e.g., [Ritschel et al. 09a, Wyman and Nichols 09, Kapalanyan 09, Dachsbacher and Stamminger 06, Dachsbacher and Stamminger 05]) that describe methods for indirect one-bounce

illumination for fully dynamic scenes, but they do not account for indirect shadows. Only a handful of methods for indirect illumination have been described to date that also include support for indirect shadows in the context of fully dynamic scenes and interactive frame rates (e.g., [Ritschel et al. 08, Ritschel et al. 09a, Ritschel et al. 09b, Kapalanyan and Dachsbacher 10, Yang et al. 09, Thibieroz and Gruen 10]).

Direct3D 11-capable GPUs allow the concurrent construction of linked lists using scattering writes and atomic operations (see [Yang et al. 09]). This capability is used as the basic building block for the solution to real-time indirect shadowing described in this chapter. Linked lists open the door for a new class of real-time algorithms to compute indirect shadows for fully dynamic scenes using ray-triangle intersections. The basic idea behind these techniques is to dynamically build data structures on the GPU that contain lists of triangles that represent low *level-of-detail* (LOD) versions of potential blockers of indirect light. Most game engines already rely on having low LOD versions of game objects for rendering or simulation purposes. These low LOD objects can readily be used as the approximate blockers of indirect light, as long as the LOD is good enough to capture the full topology of objects for proper self-shadowing.

The data structures containing lists of triangles are traversed using ray tracing to detect if some amount of the indirect light is blocked. Although this approach could probably be used to implement ray tracing of dynamic scenes in general, the following discussion considers only the application of linked lists in the context of the computation of indirect shadows and for low LOD-blocker geometry.

[Thibieroz and Gruen 10] discuss some of the implementation details of a proof-of-concept application for the indirect shadowing technique presented in [Yang et al. 09]. However, the scene used to test the indirect illumination solver did not contain any dynamic objects. Tests with more complicated dynamic scenes and rapidly changing lighting conditions revealed flickering artifacts that are not acceptable for high-quality interactive applications. The algorithms presented below address these issues and are able to deliver real-time frame rates for more complicated dynamic scenes that include moving objects and changing lighting conditions.

As described in the overview, the techniques explained below separate the process of computing indirect illumination into three phases. The reason for specifically separating the computation of indirect light and blocked indirect light is that it makes it easier to generate the blocked indirect light at a different fidelity or even at a different resolution than the indirect light. Furthermore, game developers will find it easier to add just the indirect light part of the technique if they can't rely on Direct3D 11-capable hardware for indirect shadowing. The three phases for computing the full one-bounce illumination approximation are explained in detail in the following sections.

3.3 Phase 1: Computing Indirect Illumination without Indirect Shadows

The approach for one-bounce indirect illumination described in this chapter is based on "Reflective Shadow Maps" [Dachsbacher and Stamminger 05].

One starts by rendering a G-buffer as seen from the eye and a reflective shadow map (RSM) as seen from the light. RSMs use the observation that one-bounce indirect illumination from a light source is caused by surfaces that are visible from the light's viewpoint.

As a RSM is also essentially a G-buffer of the scene as seen from the light, each texel of the RSM can be treated as a *virtual point light* source (VPL). So for each screen pixel of the camera-view G-buffer, one accumulates indirect light from a kernel of RSM texels as shown in Figure 3.1.

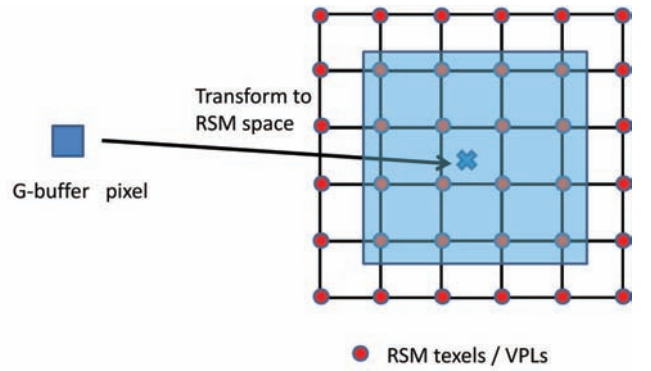
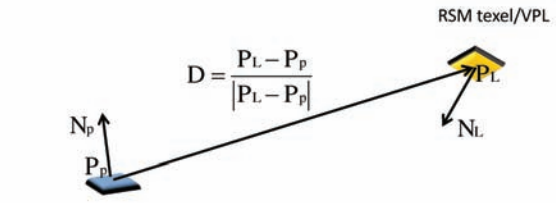


Figure 3.1. Accumulating light from a kernel of RSM texels.

The contribution of each VPL is computed as shown in Figure 3.2. If possible the approximate surface area that each RSM pixel represents should be rendered to the RSM as well.

[Thibieroz and Gruen 10] use a well-known trick to achieve good quality and real-time frame rates for a 20×20 kernel of VPLs. They employ a form of bilinear filtering. This works nicely for a static scene and a low-resolution RSM. It turns out though, that for dynamic scenes a much higher-resolution RSM ($\geq 512 \times 512$) and also a much bigger kernel of RSM pixels ($\geq 80 \times 80$) are needed. For dynamic scenes, the techniques described in [Thibieroz and Gruen 10] do not provide real-time frame rates.

One way to reach good performance for a VPL kernel of, for example, 81×81 pixels is to not use a full kernel of VPLs at each screen pixel, but to use a dithered pattern of VPLs that considers only one out of $N \times N$ VPLs. The samples for each screen pixel are then offset by its 2D pixel position modulo N. This is



$$\text{Contribution}_{VPL} = \frac{\text{saturate}(N_p \cdot D) \cdot \text{saturate}(N_L \cdot (-D))}{|P_L - P_p|^2} \cdot \text{Col}_{VPL}$$

Figure 3.2. Computing the contribution of a VPL.

similar in spirit to [Segovia et al. 09] but does not actually split the G-buffer into sub-buffers for better memory coherence.

The shader in Listing 3.1 demonstrates how this can be implemented for a pattern that only uses one out of 6×6 VPLs.

```
// this function evaluates the weighting factor
// of a vpl
float evaluateVPLWeightingFac(RSM_data d, // data for VPL
                              float3 f3CPos, // pos of pix
                              float3 f4CN, // normal of pix
                              )
{
    // compute indirect light contribution weight
    float3 f3D = d.f3Pos.xyz - f3CPos.xyz;
    float fLen = length( f3D );
    float fInvLen = rcp( fLen );
    float fDot1 = dot( f3CN, f3D );
    float fDot2 = dot( d.f3N, -f3D );

    return  saturate( fInvLen * fInvLen ) *
            saturate( fDot1 * fInvLen ) *
            saturate( fDot2 * fInvLen );
}

// this function computes the indirect light from a kernel of
// VPLs inside the RSM.A repetitive screen space pattern is used
// to do interleaved shading to reduce the number
// of samples to look at
float3 computeIndirectLight(
                            float2 tc, // RSM cords of g-bug pix
                            int2 i2Off,// kernel start offset
```

```
                                float3 f3CPos, // g-buf pix pos
                                float3 f3CN // g-buf pix normal )
{
    float3 f3IL = (0.0f).xxx; // init indirect illumination

    // loop over VPL kernel
    for( float row = -LFS; row <= LFS; row += 6.0f )
    {
        for( float col = -LFS; col <= LFS; col += 6.0f )
        {
            // unpack RSM g-buffer data for VPL
            RSM_data d = LoadRSMData( tc, i2Off, row, col );

            // accumulate weighted indirect light
            f3IL += d.f3Col *
                    evaluateVPLWeightingFac( d, f3CPos, f3CN ) *
                    d.PixelArea;
        }
    }

    return f3IL;
}

// indirect light is computed for a half width/ half height
// image
float4 PS_RenderIndirectLight( PS_SIMPLE_INPUT I ) : SV_TARGET
{
    // compute screen pos for RT that is 2*w, 2*h
    int3 tc = int3( int2( I.vPos.xy ) << 1, 0 );

    // start offset of the VPL kernel repeats every 6x6 pixels
    int2 i2Off = ( int2( I.vPos.xy ) % (0x5).xx );

    // load gbuffer data at the current pixel
    GBuf_data d = LoadGBufData( tc );

    // transform world space pos to rsm texture space
    float2 rtc = transform2RSMSpace( d.f3CPos );

    // compute indirect light
    float3 f3IL = computeIndirectLight( rtc, i2Off,
                                        d.f3CPos, d.f3CN );

    return float4( f3IL, 0.0f );
}
```

Listing 3.1. Accumulating indirect light over a dithered kernel of VPLs.

Note that the shader assumes that the indirect light is computed at a resolution that is half as high and half as wide as the screen resolution.

As the dithered result from Listing 3.1 is not smooth, a bilateral blurring step

Figure 3.3. Demo scene without indirect illumination.

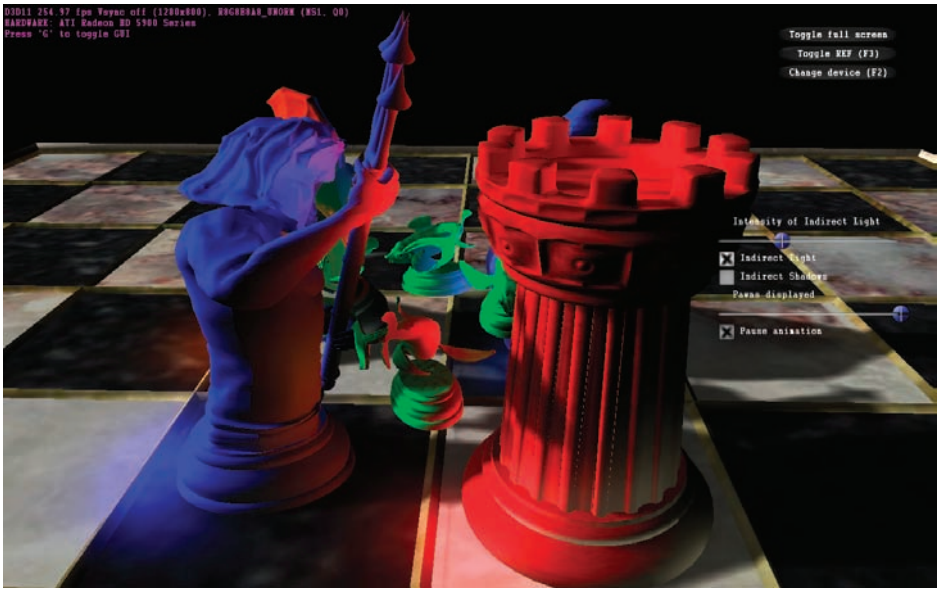


Figure 3.4. Demo scene with indirect illumination.

(see e.g., [Tomasi and Manduchi 98]) is performed and then the image is sampled up to the full-screen resolution using *bilateral upsampling* [Sloan et al. 09]. Both bilateral filter operations use the differences in normal and light space-depth between a central G-buffer pixel and samples in the filter footprint.

Figures 3.3 and 3.4 show screenshots of a demo implementing the algorithm described above. The demo uses a 512×512 RSM and a dithered 81×81 kernel of VPLs. The frame rate of the demo is usually above 250 frames per seconds on an AMD HD5970 at 1280×800, which shows that the technique works fast enough to be used in interactive applications and computer games.

3.4 Phase 2: Constructing a 3D Grid of Blockers

In order to compute indirect shadows in the context of the RSM-based indirect illumination described above, one ideally needs to find a way to detect which VPLs can really be seen from a point in space and which are occluded by blocker geometry. Conceptually, the most straightforward way to detect occluding geometry is to trace rays from the position of a G-buffer pixel to the VPLs in the RSM.

As this has to work for arbitrary and ideally dynamic scenes a method needs to be found to quickly build a data structure that allows for fast ray tracing of these scenes. A simple solution to this problem is to dynamically update a 3D grid with the relevant blocker geometry (see e.g., [Thibieroz and Gruen 10]). Similar in spirit, a 3D grid containing a list of blocker triangles per cell can be constructed on Direct3D 11 hardware using a scattering compute shader. [Yang et al. 09] describes how to concurrently build linked lists on a GPU, and the exact same principles are used to generate a 3D grid with a list of triangles in each grid cell. Please note that only the triangles of a very low LOD representation of blocker objects need to be added to the 3D grid. Usually even complex blocker objects can be approximated sufficiently with only a few hundred triangles in order to generate reasonable indirect shadows.

A compute shader is used to directly pull triangle data from the index and vertex buffers of the blocker objects and to perform a rough and conservative 3D rasterization of several hundred blocker triangles in parallel. Each triangle is added to the list of triangles of each grid cell it touches. This process is depicted in Figure 3.5 and is run every frame to captures any change in position, orientation, or animation pose of the blocker geometry.

The simplest rasterization loop that can be implemented is to add a blocker triangle to all cells its bounding box touches. As intersection tests do carry out full ray-triangle intersections, this overly conservative rasterization doesn't create any false occlusion and will work well if the blocker triangles are not a lot larger than a cell of the 3D grid. As soon as triangles get more than twice as big as the grid cells, this form of rasterization becomes inefficient. Similarly, the

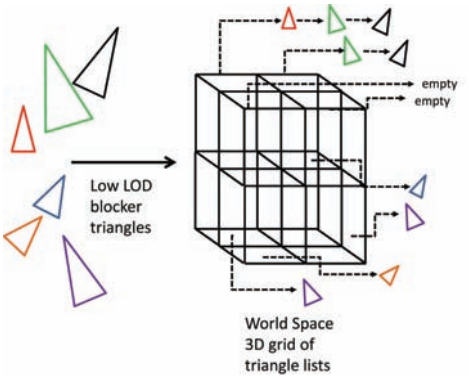


Figure 3.5. Rasterizing blocker triangles into a 3D grid.

effectiveness of the grid for reducing the number of ray-triangle intersections is reduced if blocker triangles are a lot smaller than the cells of the grid.

The following compute shader implements the simple rasterization just described.

```
//
// Add triangles defined by an index and a vertex buffer into
// the linked list of each 3D grid cell they touch
//
// Note: This is a simplified 3D rasterization loop as it
// touches all grid cells that are touched by the bounding box
// of the triangle and adds the triangle to the list of all
// these grid cells
//
[numthreads( GROUPSIZE, 1, 1 )]
void CS_AddTrisToGrid(  uint3 Gid : SV_GroupID,
                        uint3 GTid : SV_GroupThreadID,
                        uint GI : SV_GroupIndex )
{
    // compute offset into index buffer from the thread and
    // group ids
    uint uOffset = GROUPSIZE * Gid.y + GTid.x;
    uint3 indices;
    LinkedTriangle t;
    uint3 start, stop;

    // only process valid indices
    if( uOffset < g_IndexCount ) {
        // add startindex to the offset
        uOffset += g_StartIndexLocation;

        // fetch three indices for triangle
```

```
        Indices = fetchIndices( uOffset );

        // add base vertex location
        indices += g_BaseVertexLocation.xxx;

        // compute offset for vertices into the vertex buffer
        uint3 voffset = indices * g_VertexStride.xxx +
                            g_VertexStart.xxx;

        // load vertex data of triangle--prepare triangle
        float3 v0 = g_bufVertices.Load( voffset.x ).xyz;
        float3 v1 = g_bufVertices.Load( voffset.y ).xyz;
        float3 v2 = g_bufVertices.Load( voffset.z ).xyz;

        // now call e.g., skinning code for the vertices
        // if the vertices belong to a skinned object

        t.v0 = v0;
        t.edge1 = v1 - t.v0;
        t.edge2 = v2 - t.v0;

        // compute bounding box of tri and start and
        // stop address for rasterization
        computeStartStop( start, stop, v0, v1, v2 );

        // iterate over cells
        for( uint zi = start.z; zi <= stop.z; ++zi ) {
            for( uint yi = start.y; yi <= stop.y; ++yi ) {
                for( uint xi = start.x; xi <= stop.x; ++xi ) {
                    // alloc new offset
                    uint newOffset = LinkedTriGridBuffer.
                                        IncrementCounter();
                    uint oldOffset;

                    // update grid offset buffer
                    StartOffsetBuffer.
                    InterlockedExchange( 4 * ( xi + yi *
                                        CELLS_XYZ + zi *
                                        CELLS_XYZ * CELLS_XYZ ),
                                        newOffset, oldOffset );

                    // store old offset
                    lt.prev = oldOffset;

                    // add triangle to the grid
                    LinkedTriGridBuffer[ newOffset ] = t;
} } } } }
```

Listing 3.2. Rasterizing blocker triangles into a 3D grid of lists.

3.5 Phase 3: Computing the Blocked Portion of Indirect Light

Now that the grid has been created, it can be used to detect which VPLs are occluded and which are not occluded. In a separate rendering pass, the light from all blocked VPLs is accumulated. The reason for not interleaving this pass with phase 1 is that this pass generates more flexibility in terms of how many rays are shot for detecting blocked indirect light. Shooting rays is still a costly operation and one typically wants to shoot only few.

Listing 3.3 shows the shader code for accumulating blocked indirect light for a half-width, half-height buffer and a dithered 13×13 kernel of indirect light. As an optimization, rays are shot only if there is some amount of indirect light at the current pixel.

Note that the demo implementation uses an adapted version of the fast ray-triangle intersection algorithm presented in [Möller and Trumbore 97].

```
//
// This function walks the 3D grid to check for intersections of
// triangles and the given edge
//
float traceRayLinkedTris( float3 f3OrgP, float3 f3D )
{
        float fIntersection = (0.0f), fI, fLen;
        float3 f3Inc, f3P;

        // setup the march along the ray trough the grid cells
        setupRay( fLen, f3P, f3Inc );

        // do the march
        for( fI = 0.0f;
             fI <= fLen;
             fI += 1.0f, f3P += f3Inc )
        {
            // check_for_intersection walks through the list
            // of tris in the current grid cell and computes
            // ray triangles intersections
            if( check_for_intersection( int3( f3P ), f3P,
                                f3OrgP, f3D ) != 0.0f ) {
                fIntersection = 1.0f;
                break;
            }
        }
    return fIntersection;
}

float3 computeBlockedIndirectLight( float2 tc, float2 fc,
                                    int2 i2Off, float3 f3CPos,
                                    float3 f3CN )
```

```
{
    float3 f3IL = (0.0f).xxx;

    // loop over VPL kernel
    for( float row = -SFS; row <= SFS; row += 6.0f) {
        for( float col = -SFS; col <= SFS; col += 6.0f ) {
            // unpack RSM g-buffer data for VPL
            RSM_data d = LoadRSMData( adr );

            // compute weighting factor for VPL
            float f = evaluateVPLWeightingFac( d,f3CPos, f3CN );

            if( f > 0.0f ) {
                f3IL += traceRayLinkedTris(f3CPos.xyz, f3D )*
                        * d.f3Col * f;

            }
        }
    }

    // amplify the accumulated blocked indirect light a bit
    // to make indirect shadows more prominent
    return 16.0f * f3IL;
}

// renders the accumulated color of blocked light using a 3D
// grid of triangle lists to detect occluded VPLs
float4
PS_RenderBlockedIndirectLightLinkedTris( PS_SIMPLE_INPUT I ) :
SV_TARGET
{
    int3 tc = int3( int2( I.vPos.xy ) << 1, 0 );
    int2 i2Off = ( int2( I.vPos.xy ) % (0x5).xx );
    Gbuf_data d = LoadGBufData( tc );

    // transform world space pos to rsm texture space
    float2 rtc = transform2RSMSpace( d.f3CPos );

    float3 f3IS = 0.0f;
    float3 f3IL = g_txIndirectLight.SampleLevel(
                  g_SamplePointClamp, I.vTex.xy, 0 ).xyz;

    // is any indirect light (phase 1) reaching this pixel
    [branch]if( dot( f3IL, f3IL ) > 0.0f )
        f3IS = computeBlockedIndirectLight( rtc, i2Off,
                                            d.f3CPos, d.f3CN );

    return float4( f3IS, 0.0f );
}
```

Listing 3.3. Accumulating blocked indirect light.

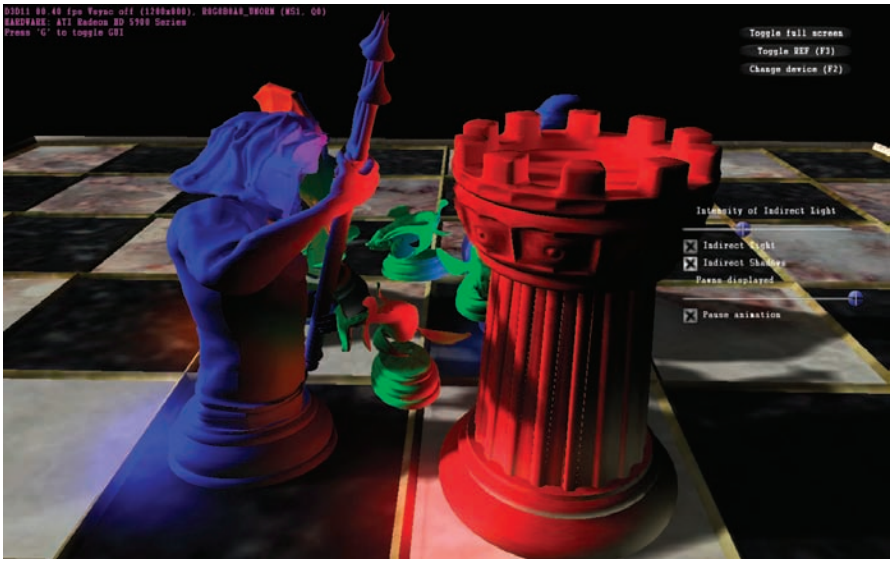


Figure 3.6. Demo scene with indirect illumination and indirect shadows.

Again, the dithered, blocked indirect light is blurred and upsampled using a bilateral filter. After that, the blocked indirect light is subtracted from the indirect light, and the result is clamped to make sure that indirect illumination doesn't become negative. This generates the full indirect illumination approximation with indirect shadowing. Finally, indirect illumination is combined with direct illumination and shadowing to produce the final image as shown in Figure 3.6.

The performance for rendering the full one-bounce indirect illumination, including indirect shadows with tracing nine rays per pixel is at 70–110 fps for a $32 \times 32 \times 32$ grid and a resolution of 1280×800 on an AMD HD5970. The number of blocker triangles that get inserted into the 3D grid in every frame is in the order of 6000.

3.6 Future Work

There are several future directions for improving the techniques described in this chapter.

1. The use of a hierarchical grid for speeding up ray-triangle intersections.
2. The insertion of references to a hierarchical structure, for example., a kd-tree (encoded in a buffer) into the lists of each grid cell. This would allow for faster ray-tracing of rigid or static scene elements.

3. The use of a binary 3D grid that could be generated by a scattering pixel shader and the use of the SM5 instruction for an atomic binary or operation (`InterlockeOr()`). Early experiments show that this is feasible and fast, but the resulting blocked indirect light is not stable for low enough resolutions of the binary 3D grid.

4. Instead of computing the accumulated contribution of blocked VPLs for each pixel, it would be possible to compute a spherical harmonics projection of the blocked indirect light of a distribution of VPLs at the center of each cell of the 3D grid. For a given screen pixel one could reconstruct a smooth approximation of the blocked indirect light from the tri-linear interpolation of eight sets of the spherical harmonics coefficient of the eight relevant grid cells.

Bibliography

[Dachsbacher and Stamminger 05] Carsten Dachsbacher and Marc Stamminger. "Reflective Shadow Maps. " In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pp.203–231. New York, ACM, 2005.

[Dachsbacher and Stamminger 06] Carsten Dachsbacher and Marc Stamminger. "Splatting Indirect Illumination." In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pp. 93–100. New York, ACM, 2006.

[Eisemann and Décoret 06] Elmar Eisemann and Xavier Décoret. "Fast Scene Voxelization and Applications." In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pp. 71–78. New York, ACM, 2006.

[Kapalanyan 09] Anton Kapalanyan. "Light Propagation Volumes in CryEngine 3." In *Advances in Real-Time Rendering in 3D Graphics and Games Course - SIGGRAPH 2009*. Available online http://www.crytek.com/cryengine/cryengine3/presentations/light-propagation-volumes-in-cryengine-3, 2009.

[Kapalanyan and Dachsbacher 10] Anton Kaplanyan and Carsten Dachsbacher. "Cascaded Light Propagation Volumes for Real-Time Indirect Illumination," In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pp. 99–107. New York: ACM, 2010.

[Möller and Trumbore 97] Thomas Möller and Ben Trumbore. "Fast, Minimum Storage Ray/Triangle Intersection," *journal of graphics tool* 2:1 (1997), 21–28.

[Ritschel et al. 08] T. Ritschel, T. Grosch, T. M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz. "Imperfect Shadow Maps for Efficient Computation of Indirect Illumination." *ACM Trans. Graph.* 27:5 (2008), 129:1–129:8. .

[Ritschel et al. 09a] T. Ritschel, T. Grosch, and H.-P. Seidel. "Approximating Dynamic Global Illumination in Image Space." In *I3D '09: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pp. 75–82. New York, ACM, 2009.

[Ritschel et al. 09b] T. Ritschel, T. Engelhardt, T. Grosch, H.-P. Seidel, J. Kautz, and C. Dachsbacher. "Micro-Rendering for Scalable, Parallel Final Gathering." *ACM Transactions on Graphics (Proc. SIGGRAPH Asia 2009)* 28:5 (2009), 132:1–132:8.

[Sloan et al. 09] P.-P Sloan, N. Govindaraju, D. Nowrouzezahrai and J. Snyder. "Image-Based Proxy Accumulation for Real-Time Soft Global Illumination." In *Proceedings of the 15th Pacific Conference on Computer Graphics*, pp. 97–105. Washington, DC: IEEE Computer Society, 2009.

[Segovia et al. 09] B. Segovia, J.C. Iehl R. Mitanchey and B. Péroche. "Non-Interleaved Deferred Shading of Interleaved Sample Patterns." in *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pp. 53–60. Aire-la-Ville, Switzerland, Eurographics Asociation, 2009.

[Tomasi and Manduchi 98] C. Tomasi and R. Manduchi. "Bilateral Filtering for Gray and Color Images." In *Proceedings of the Sixth International Conference on Computer Vision*, pp. 839–846. Washington, DC: IEEE Computer Society, 1998.

[Thibieroz and Gruen 10] N. Thibieroz and H. Gruen. "'OIT and Indirect Illumination Using DX11 Linked Lists." In *Proceedings of Game Developers Conference*, 2010.

[Wald et al. 09] I. Wald. T. Ize, A. Kensler, A. Knoll, and S¿ G¿ Parker. "Ray Tracing Animated Scenes Using Coherent Grid Traversal." *ACM Transanction on Graphics (Proc. SIGGRAPH ASIA)* 25:3 (2006), 485–493.

[Wyman and Nichols 09] C. Wyman and G. Nichols. "Multiresolution Splatting for Indirect Illumination." In *I3D '09: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, pp. 83–90. News York: ACM, 2009.

[Yang et al. 09] J. Yang, J. Hensley, H. Gruen, and N. Thibieroz. "Dynamic Construction of Concurrent Linked-Lists for Real-Time Rendering." *Computer Graphics Forum (Eurographics Symposium on Rendering 2010)*29:4 (2010), 1297–1304.

4

III

Real-Time Approximation of Light Transport in Translucent Homogenous Media

Colin Barré-Brisebois and Marc Bouchard

4.1 Introduction

When reproducing visual elements found in nature, it is crucial to have a mathematical theory that models real-world light transport. In real-time graphics, the interaction of light and matter is often reduced to local reflection described by *bidirectional reflectance distribution functions* (BRDFs), for example, describing reflectance at the surface of opaque objects [Kurt 09]. In nature, however, many

Figure 4.1. A partially translucent statue of Athena.

Figure 4.2. The final result.

objects are (partly) translucent: light transport also happens within the surface (as shown in Figure 4.1).

To simulate light transport inside objects in real time, developers rely on various complex shading models and algorithms, (e.g., to replicate the intricate subsurface scattering found in human skin [d'Eon 09,Hable 09].) Conversely, this article presents a fast real-time approximation of light transport in translucent homogeneous media, which can be easily implemented on programmable graphics hardware (PC, as well as video game consoles). In addition, the technique scales well on fixed and semi-fixed pipeline systems, it is artist friendly, and it provides results that fool the eyes of most users of real-time graphics products. This technique's simplicity also permits fast iterations, a key criteria for achieving visual success and quality in the competitive video game industry. We discuss the developmental challenge, its origins, and how it was resolved through an initial and basic implementation. We then present several scalable variations, or improvements to the original technique, all of which enhance the final result (see Figure 4.2).

4.2 In Search of Translucency

First and foremost, the technique we present was originally implemented in the graphics research pursued for EA Montréal's *Spore Hero* video game. In this game, we wanted to create simple-yet-eerie, translucent, mushroom-like worlds. Our materials were inspired by the statue in Figure 4.1 and had to demonstrate convincing diffuse translucency, whereby the amount of light that traveled inside

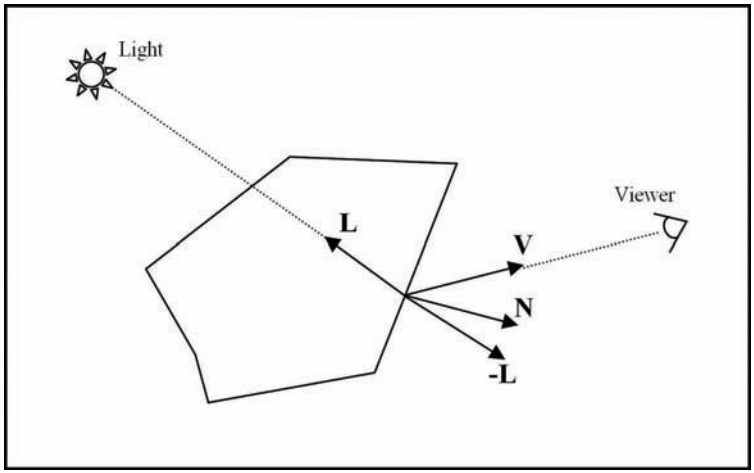


Figure 4.3. Lighting vectors.

the shape was influenced by the varying thickness of that same shape. In mathematical terms, this means that the amount of light that penetrates the surface of the shape (but also the amount of light exiting at the opposite hemisphere to the BRDF) can be defined with a *bidirectional transmittance distribution function* (BTDF). Our current method attempts to phenomenologically replicate inner-surface diffusion, in which light traveling inside the object is scattered based on material properties. This phenomena can be described using a *bidirectional surface scattering reflectance distribution function* (BSSRDF) [Jensen 01]. With this technique, we wanted to approximate the BSSRDF using minimal available GPU resources. After careful analysis, we discovered that using distance-attenuated regular diffuse lighting, combined with the distance-attenuated dot product of the view vector and an inverted light vector, would simulate basic light transport inside an object (see Figure 4.3).

Through this research, executed in prototype levels not included with the final product, we discovered that the technique worked well for numerous simple shapes. However, the technique did not take thickness into account and we had to return to the drawing board, seeking a solution that was more effective and complete.

4.3 The Technique: The Way Out is Through

Using the aforementioned math, it is possible to create a rough simulation of highly scattering material that works well for very simple objects, such as spheres and cubes, since most lights used at runtime show radial diffusion properties. In

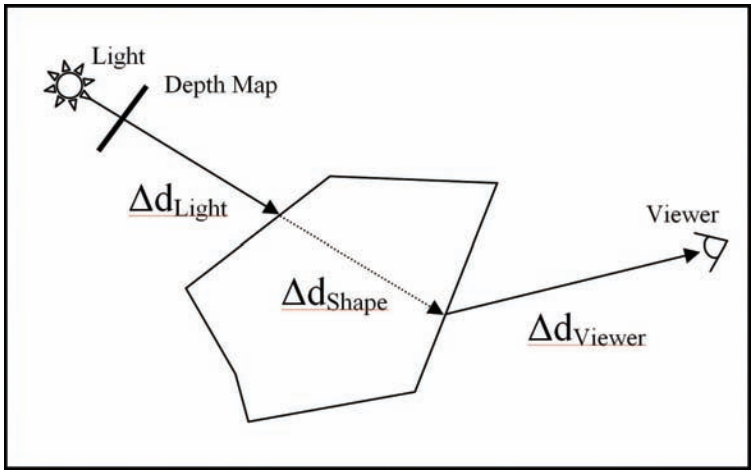


Figure 4.4. Local thickness for Hebe.

cases with more complex models, it is necessary to be aware of the thickness, or rather, the distance traveled by the light inside the shape, in order to properly compute light transport inside that same shape. As seen in [Dachsbacher 03, Green 04, Ki 09], this thickness can be computed using depth maps. The referential, as determined by the depth map, allows us to easily compute the distance traveled by the light from the light source Δd_{Light}, through the shape Δd_{Shape}, and to the pixel Δd_{Viewer} (see Figure 4.4).

Despite the fact that this method provides quite convincing results, successfully creating a technique that achieves a similar effect on current gaming platforms, without relying on depth maps would be beneficial. It would eliminate the need for an additional depth rendering pass, wherein the geometry is submitted a second time (and possibly a third time, in cases where it was already submitted to a global shadowing solution). The method is acceptable in cases where the object rendered with translucency uses its own depth map for shadows because its memory and performance cost are already amortized. Our team wanted to expand upon this idea to find an alternative, seeking a technique that would scale well on a variety of programmable and semifixed pipeline graphics hardware without relying on depth maps. In doing so, we wanted to test the limits and avoid detrimental changes to the runtime. To accomplish this task, our team needed to find the means to establish object thickness, or rather, areas on the object that should be translucent or opaque: we define this information as *local thickness*.

4.3.1 Variation 1: Computing Local Thickness

As seen in [Sousa 08], it is possible for artists to define a texture in which the values are approximately representative of the mesh's thickness; with dark values for opaque, and bright values for translucent. Effectively demonstrated in *Crysis*, this method works well for semiflat surfaces such as tree leaves. Unfortunately, in cases where the environment has numerous translucent objects, shaped in various forms, the process of defining which areas on the shape are translucent is a tedious manual process.

To streamline this process we rely on a normal-inverted and color-inverted computation of *ambient occlusion* (AO), which can be done offline and stored in a texture using your favorite modeling software. Since ambient occlusion determines how much environmental light arrives at a surface point, we use this information for the inside of the shape. Through inverting the surface normal during the computation, it is possible to find an approximate result that tells us how much light traveling inside the homogenous media would become occluded at the point where it exits the shape.

The result is a texture with inverted colors that depict two types of area, translucent areas in white and opaque areas in black (see Figure 4.5). Conversely, in cases where you have a good level of tessellation, it is possible to store this information in the vertex color. Finally, we can use this information to improve the computation of light transport, where the final result lies between real

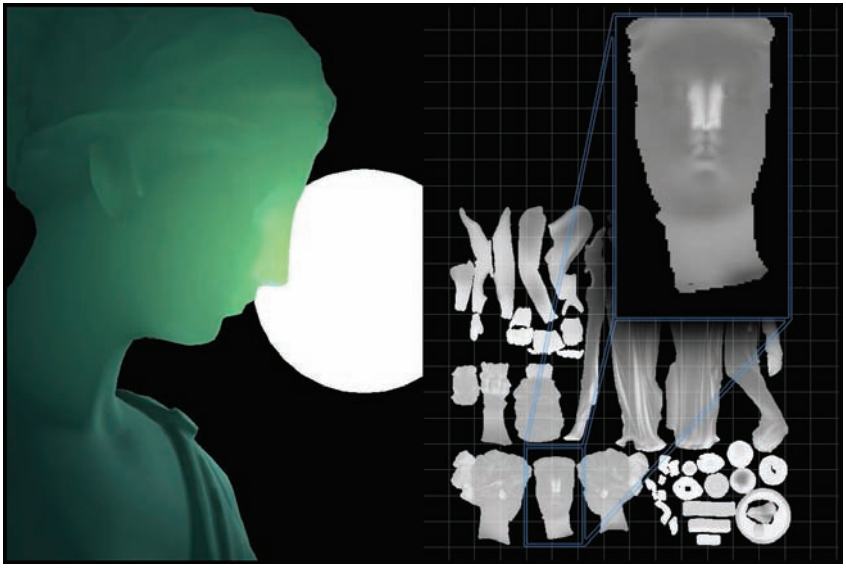


Figure 4.5. Local thickness for Hebe.

subsurface scattering and distance-based attenuation. The inverted AO gives a feeling of scattering, (i.e., collecting light), while using a single distance value for distance-based attenuation.

```
// fLightAttenuation == Light attenuation from direct lighting
// cBRDF             == Computed color for BRDF lighting
// cDiffuse          == Light Diffuse Color
// cTranslucent      == Light Translucency Color

// Compute the BSSRDF
float fLTDot = pow(saturate(dot(vCamera, vLight)), fLTPower);
float fLT = fLightAttenuation * tex2D(texInvAO, input.vUV);
fLT += fLTDot * fLightAttenuation * fLTScale;

// Compute the final color for the BSSRDF (translucency)
float3 cBSSRDF = lerp(cDiffuse, cTranslucent, fLT) * fLT;

// The final result
return float4(cBRDF + cBSSRDF, 1.0f);
```

Listing 4.1. The light transfer in HLSL.

4.3.2 Variation 2: Screen-Space Thickness

In cases where you can afford the extra computation, and still be below the cost of using depth maps, an alternative improvement provided by this new technique relies on a screen-space representation of thickness. As seen in [Oat 08], per-pixel thickness can be easily approximated in an additional pass using a blend-mode trick. Once this alternative to local thickness is computed, it is possible to scale the simplified light-transport result based on the angle of the camera to the light source.

This alternative must be used wisely because it is not valid for every scenario. Given that the information originates from the camera, rather than from the light, it is possible to know only how thick the foreground surface is. Nonetheless, this added information allows us to improve the final result and can be used in conjunction with the local thickness.

There might be game scenarios in which you could use screen-space thickness as the sole means of achieving convincing translucency, and in which there is no need for precomputed local thickness. For example, one could imagine a scenario in which there is a light far in the distance (i.e., in a tunnel) and there are numerous objects that rotate and flow toward the viewer. This scenario is one example where screen-space thickness can provide positive results, especially when applied to arbitrary and animated shapes. As shown in Figure 4.6, screen-space thickness is also effective in cases where there is a bright hemispheric light

Figure 4.6. Using screen-space thickness only.

surrounding the objects. Unfortunately, a still image does not do this variation justice, therefore we recommend viewing the demo on the website..

4.3.3 Variation 3: Improving Local Surface Variation

As a further enhancement, one can distort the inverted light vector (see Figure 4.3) based on the surface normal to improve the final result by showing local surface variation. This improvement can also be made using an additional noisy normal map that is customized for the surface type. A single sample is enough, however, if more samples are provided, the final result will be improved. This approach also works quite well if the object's surface is porous (as shown in Figure 4.2).

An example of the aforementioned variation is provided on the accompanying CD. Obviously, it is a quick alternative which does not measure up to the methods of [d'Eon 09, Hable 09] when simulating realistic human skin. However, it can certainly be used as a ready substitute for fast subsurface scattering on objects such as the ears and nose, various animated and static environmental shapes, and even cartoon/nonhuman characters. Further, the image shown in Figure 4.7 relies only on the distortion, not on precomputed local thickness. Combined, both variations significantly increase the quality of the final result (see Figure 4.9).

4.4 Performance

The performance numbers for all techniques illustrated in this paper are given in Table 4.1. Timings are given in *frames-per-second* (FPS) and the complex-

Figure 4.7. Without (left) and with (right) normal distortion.

ity is presented in terms of added *arithmetic logic unit* (ALU) and *texture unit* (TEX) shader instructions. These benchmarks were established using an NVIDIA GeForce GTX260, at a resolution of 1280×720, in a scene comprised of several instances of our Hebe statue, with approximately 200,000 triangles.

Overall, these numbers illustrate that our technique approximates light transport inside translucent homogenous media at a very reasonable cost. This method also provides significant benefits in cases when developers want a quick impression of subsurface scattering. In fact, the technique requires only 17 additional instructions to achieve a significant and convincing effect. Finally, if the option is financially viable, adding screen-space thickness to the computation will further improve the final result. The cost of this computation can be managed through the use of a quarter-sized buffer, with filtering in order to prevent artifacts at the boundaries. Realistically, one could rely on a lightweight separable-Gaussian

Technique	**FPS**	**Instructions (ALU+TEX)**
Without Translucency	1100	-
Variation 1: Translucency + Local Thickness	1030	17 (16 + 1)
Variation 2: Translucency + Screen-Space Thickness	740	20 (17 + 3)
Variation 3: Translucency + Normal Distortion	1030	19 (18 + 1)

Table 4.1. Performance numbers.

blur, and even use a bilateral filter when up-sampling. A blurred result would definitely complement our diffuse computation.

4.5 Discussion

In the following section we discuss various elements and potential concerns that were not forgotten but were put aside during the development of this technique and its variations. These concerns will now be addressed.

4.5.1 Caveat? Best-Case Scenario?

Because this technique is an approximation, some specific instances will not yield optimal results. Figure 4.8 shows an example in which light travels in and out of the concave hull at point **A**, casting a shadow on point **B**. Though it is possible to generate local thickness that is aware of the concavity (by changing the parameters used for the ambient occlusion computation in Section 4.3.1), this has to be minimized when demonstrating visually convincing examples of diffuse translucency, as is effectively represented in Figure 4.5 and Figure 4.9. Our technique works more effectively with convex hulls or hulls with minimal concavity.

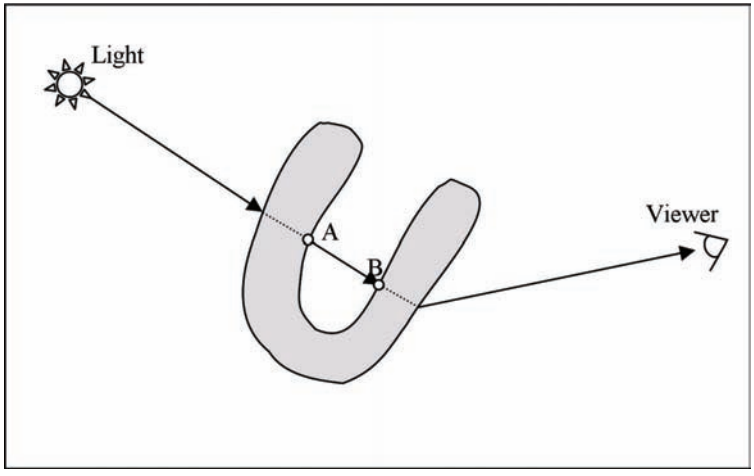


Figure 4.8. Concave hull.

4.5.2 Deferred Rendering?

Rendering engines are evolving toward a deferred shading model. Thus, this technique must be adapted accordingly. Given that deferred implementations

differ from one project to the next, it is pertinent to provide general hints regarding the adaptation of the technique within a deferred context. These hints are an adequate starting point, leaving clear room for improvement. An example implementation is provided in the web materials.

The implementation is dependent on available space on the G-buffer. In cases where a single channel is available, the local thickness can be stored as a grayscale value: this is done in the same way that it is stored for specular maps. Subsequently, the light and view-dependent part of the BSSRDF computation can be processed at the same time as the deferred lighting pass. The final result can then be combined with the scene at the same time that the diffuse lighting is computed. In cases where there is no space left for storing the local thickness, this technique will have to be treated in an additional pass and the translucent objects will have to be rendered once again. Fortunately, the z-buffer will already be full and this minimizes the number of affected pixels.

4.6 Conclusion

This article illustrates an artist-friendly, fast and scalable real-time approximation of light transport in translucent homogenous media. Our technique allows developers to improve their games' visuals by simulating translucency with reasonable and scalable impact on the runtime. Providing such convincing results through

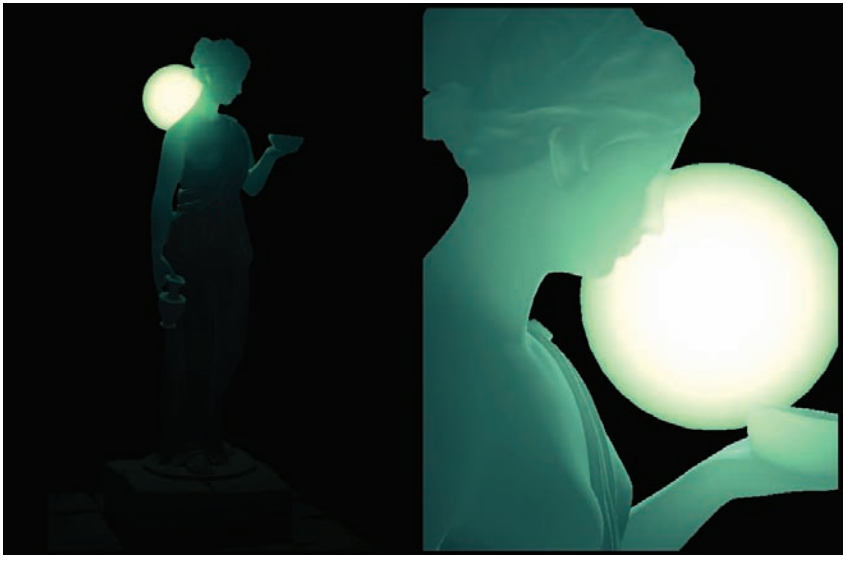


Figure 4.9. The final result, with Hebe.

simple means is essential in the development of triple-A games. We hope that this technique will inspire developers in the continuous effort to improve games by promoting the development of new techniques that carefully blend art and technology. In the end, our objective is not to focus upon mathematical perfection, but to create convincing results that push the visual boundaries of the gaming industry.

4.7 Demo

The web materials accompanying this book contain an implementation of our technique and its variations in the form of an AMD RenderMonkey sample. HLSL code is also provided in a text file and a short video demonstration of the technique is included.

4.8 Acknowledgments

We would like to thank the following individuals at Electronic Arts for reviewing this paper: Sergei Savchenko, Johan Andersson and Dominik Bauset. We are also grateful for the support of EA's Wessam Bahnassi, Frédéric O'Reilly, David "Mojette" Giraud, Gabriel Lassonde, Stéphane Lévesque, Christina Coffin and John White. Further, we would like to thank Sandie Jensen from Concordia University for her genuine and meticulous style editing. We would also like to thank our section editor, Carsten Dachsbacher, for his thorough reviews and inspiring savoir-faire in search of technical excellence through simplicity. Finally, Marc Bouchard would like to thank Ling Wen Kong and Elliott Bouchard, and Colin Barré-Brisebois would like to thank Simon Barré-Brisebois, Gene-viève Barré-Brisebois and Robert Brisebois.

Bibliography

[Hable 09] J. Hable, G. Borshakov, and J. Heil. "Fast Skin Shading." In *ShaderX*^{7}, pp. 161–173. Hingham, MA: Charles River Media, 2009.

[Dachsbacher 03] Carsten Dachsbacher and Marc Stamminger. "Translucent Shadow Maps." In *Proceedings of the 14th Eurographics Workshop on Rendering*, pp.197-201. Aire-la-Ville, Switzerland: Eurographics Association, 2003.

[d'Eon 09] Eugene d'Eon and David Luebke. "Advanced Techniques for Realistic Real-Time Skin Rendering." In *GPU Gems 3*, edited by Hubert Nguyen, pp.293–347. Reading, MA: Addison-Wesley, 2008.

[Green 04] Simon Green. "Real-Time Approximations to Subsurface Scattering." In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, edited by Randima Fernando, pp. 263-278. Reading, MA: Addison-Wesley, 2004.

[Hejl 09] Jim Hejl. "Fast Skin Shading." In *ShaderX^7: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 161–173. Hingham, MA: Charles River Media, 2009.

[Jensen 01] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. "A Practical Model for Subsurface Light Transport." In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 511-518. New York: ACM, 2001.

[Ki 09] Hyunwoo Ki. "Real-Time Subsurface Scattering Using Shadow Maps." In *ShaderX^7: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp 467–478. Hingham, MA: Charles River Media, 2009.

[Kurt 09] Murat Kurt and Dave Edwards. "A Survey of BRDF Models for Computer Graphics," *ACM SIGGRAPH Computer Graphics* 43:2 (2009), 4:1–4:7.

[Oat 08] Christopher Oat and Thorsten Christopher. "Computing Per-Pixel Object Thickness in a Single Render Pass," In *ShaderX^6: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 57–62. Hingham, MA: Charles River Media, 2008.

[Sousa 08] Tiago Sousa. "Vegetation Procedural Animation and Shading in Crysis." In *GPU Gems 3*,edited by Hubert Nguyen, pp. 373–385. Reading, MA: Addison-Wesley, 2008.

5

III

Diffuse Global Illumination with Temporally Coherent Light Propagation Volumes

Anton Kaplanyan, Wolfgang Engel, and Carsten Dachsbacher

5.1 Introduction

The elusive goal of real-time global illumination in games has been pursued for more than a decade. The most often applied solution to this problem is to use precomputed data in lightmaps (e.g., Unreal Engine 3) or precomputed radiance

Figure 5.1. Example of indirect lighting with Light Propagation Volumes in the upcoming blockbuster *Crysis 2*.

transfer (e.g., *Halo 3*). Both techniques increase the complexity and decrease the efficiency of a game production pipeline and require an expensive infrastructure (e.g., setting up a cloud for precomputation and incorporating the result into a build process).

In this chapter we describe the light propagation volumes, a scalable real-time technique that does not require a preprocess and storing additional data. The basic idea is to use a lattice storing the light and the geometry in a scene. The directional distribution of light is represented using low-order spherical harmonics. The surfaces of the scene are sampled using reflective shadow maps and this information is then used to initialize the lattice for both light propagation and blocking. A data-parallel light propagation scheme allows us to quickly, and plausibly, approximate low-frequency direct and indirect lighting including fuzzy occlusion for indirect light. Our technique is capable of approximating indirect illumination on a vast majority of existing GPUs and is battle-tested in the production process of an AAA game. We also describe recent improvements to the technique such as improved temporal and spatial coherence. These improvements enabled us to achieve a time budget of 1 millisecond per frame on average on both Microsoft Xbox 360 and Sony PlayStation 3 game consoles.[1]

5.2 Overview

The light propagation volume technique consists of four stages:

- At the first stage we render all directly lit surfaces of the scene into reflective shadow maps [Dachsbacher and Stamminger 05] (see Figure 5.2).
- Next, a sparse 3D grid of radiance distributions is initialized with the surface samples generated in the first pass (see Figure 5.3).

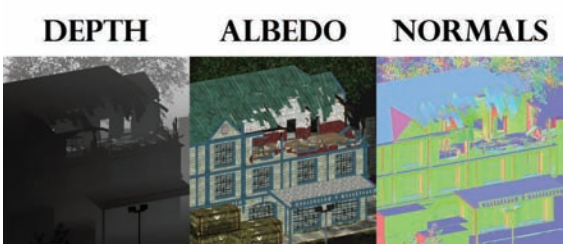


Figure 5.2. Reflective Shadow Maps store not only depth, but also information about surfaces' normals and reflected flux.

[1] *Crysis 2*, *Halo 3*, and CryENGINE 3 are trademarked. PlayStation 3, Microsoft Xbox 360, Unreal Engine 3, and Microsoft DirectX are all registered trademarks.

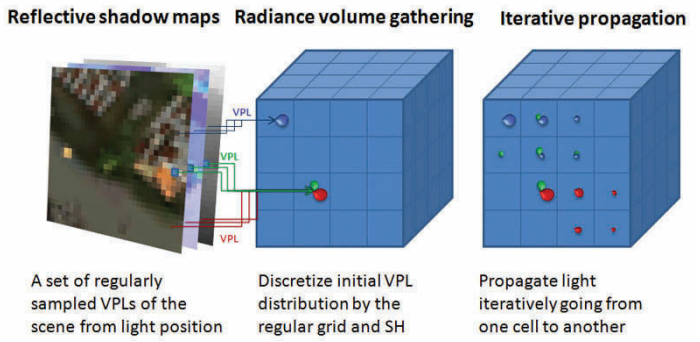


Figure 5.3. The basic steps of our method: surfaces causing one-bounce indirect illumination are sampled using RSMs, next this information is used to initialize the light propagation volumes where light is propagated, and finally this information is used to light the surfaces in the scene.

- Light is propagated in the grid using an iterative propagation scheme.
- Lastly, the scene is illuminated using the resulting grid, similar to using irradiance volumes [Tatarchuk 04].

Our light propagation volumes (LPV) technique works completely on the GPU and has very modest GPU requirements.

5.3 Algorithm Detail Description

In this section we recapitulate the core algorithm presented in [Kaplanyan and Dachsbacher 10] and emphasize several small, yet important, details. Besides the main steps of the technique we will present an important extension: cascading the reflective shadow maps. This extension allows our technique to be used in large scenes while maintaining real time performance. We also describe important general as well as platform-specific optimizations at the end of this section.

5.3.1 Reflective Shadow Maps

Reflective shadow maps (RSMs) are an extension to regular shadow maps and store not only a depth buffer, but also a normal buffer and flux buffer (Figure 5.2). It is a very fast method for sampling directly lit surfaces of a scene on the GPU, and all pixels of such an RSM can be seen as virtual light sources that generate the indirect illumination in a scene. This representation allows generating and storing samples of a scene's lit surfaces in a very efficient manner.

The size of an RSM should be chosen such that one surface sample (*surfel*) represents an area that is much smaller than one cell of the LPV to provide

Figure 5.4. A reflective shadow map captures the directly lit surfaces of the scene with a regular sampling from the light's origin.

sufficient detail in the lighting computation. We recommend an RSM size that is four times larger than the number of elements along the diagonal of an LPV (e.g., for a LPV of $32 \times 32 \times 32$, an RSM of at least 128×128 size is recommended).

As in the original approach, we assume indirect lighting from diffuse surfaces and store reflected flux in the RSM, which accounts for surface albedo as well as incident lighting (i.e., effects such as a colored projected light can be used as well).

Note that many techniques developed for shadow maps can be applied to RSMs. For example, an RSM can be cascaded like cascaded shadow maps to capture global illumination from a light source such as the sun, or it can be stored in cube maps similar to which cube shadow maps for point light sources. Figure 5.4 shows the surfels created from an RSM that is created for a directional light source.

Temporally stable rasterization. Temporal flickering can occur when an RSM frustum moves, and obviously becomes visible when the resolution of the RSM is rather low and thus scene surfaces are sampled at a coarser level. A simple solution to this problem is to move the frustum of the RSM with world-space positions snapped to the size of one texel in world space. This leads to consistently sampled points during rasterization and largely removes the sampling problems [Dimitrov 07]. If this is not possible, e.g. for perspective projections, then higher resolution RSMs are required for consistently sampling surfaces. Kaplanyan et al. [Kaplanyan 09] proposed downsampling RSMs to reduce the number of surfels for the injection stage. However, additional experiments showed that the downsampling if sometimes even slower than injecting the same number of surfaces directly into the LPV.

5.4 Injection Stage

The injection stage transforms the reflected flux of the surfels obtained from the RSMs into an initial light distribution represented using spherical harmonics (SH) stored into the LPV as spatial discretization. As we assume diffuse surfaces, the reflected flux of each surfel (neglecting if spatial extend) can be represented using a cosine lobe and thus using a low-order SH approximation. Note that we store the energy as directional intensity distribution in the LPV as described in [Kaplanyan and Dachsbacher 10].

The LPV is stored as a texture on the GPU and thus the SH approximations of the surfels can be easily accumulated into the LPV using additive blending. At this point the spatial discretization comes into play as the surfels' positions are snapped to the centers of LPV cells (see Figure 5.5). At the end of the injection stage, each cell of the LPV represents the initial radiant intensity approximated using spherical harmonics.

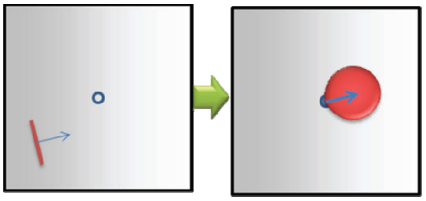


Figure 5.5. The reflected intensity distribution of a surfel obtained from an RSM is approximated using SH and snapped to the center of the closest LPV cell.

Projecting into spherical harmonics. To represent the intensity distribution of a surfel we use the first two bands of the spherical harmonics basis (i.e., four coefficients). This proved to be sufficient to represent the mostly low-frequency indirect lighting in diffuse scenes (Figure 5.6). As previously mentioned, the outgoing intensity distribution of each surfel is a (clamped) cosine lobe centered around its surface normal that is projected into the SH basis. Given the normal vector of the surfel, $\mathbf{n}$, we can obtain the SH coefficients $\mathbf{c} = (c_0, c_1, c_2, c_3)$ as [Sloan 08]:

$$
\begin{aligned}
c_0 &= \frac{\sqrt{\pi}}{2}, \\
c_1 &= -\sqrt{\frac{\pi}{3}} y, \\
c_2 &= \sqrt{\frac{\pi}{3}} z, \\
c_3 &= -\sqrt{\frac{\pi}{3}} x.
\end{aligned}
$$

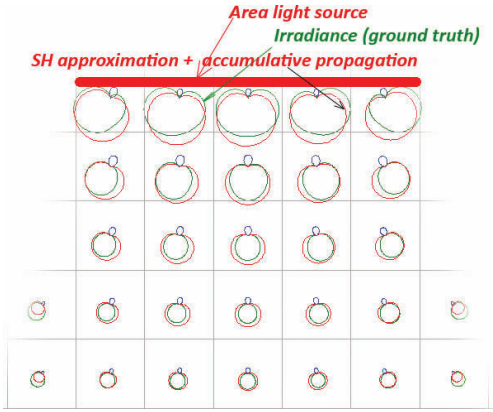


Figure 5.6. Illustration of the approximation error with 4 SH coefficients and a coarse lattice. Note that both the analytical result (green) and propagated result (red) are represented as a final convolved irradiance.

These coefficients are scaled according to the corresponding world-space size and reflected flux of the surfel, yielding four coefficients per color channel. In order to simplify the following description, we show only one set of SH coefficients.

Offsetting surfels. As the LPV is a coarse grid, we have to take care when injecting surfels, as their exact position inside a cell is no longer available after injection. If a surfel's normal points away from the cell's center, its contribution should not be added to this cell, but rather to the next cell, in order to avoid self-illumination (see Figure 5.7). For this, we virtually move each VPL by half the cell-size in the direction of its normal before determining the cell. Note that this shifting of the surfels still does not guarantee completely avoiding self-illumination and light bleeding, but largely removes artifacts from the injection stage.

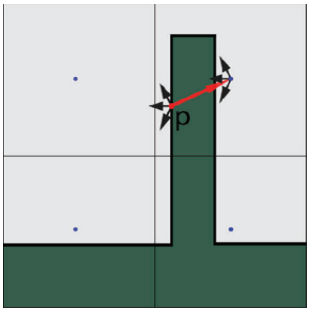


Figure 5.7. Example of a VPL injection causing self-illumination of the thin geometry.

5.4.1 Propagation

The propagation stage consists of several iterations, where every iteration represents one step of light propagation. The propagation stage has some similarity to the *SH discrete ordinate method* [Chandrasekhar 50,Evans 98]. These techniques are typically employed for light-propagation simulation in scattering media. Essentially we use the same process; however, we use a different cell-to-cell propagation scheme. The application of this method to light propagation through vacuum instead of scattering media suffers from the fact that propagation directions are blurred out. Fortunately, our results show that this is an acceptable artifact in many application scenarios.

Intensity propagation. The propagation consists of the following steps:

- The input for the first iteration step is the initial LPV obtained from the injection stage. Each cell stores the intensity as an SH-vector and we propagate the energy to the six neighbors along the axial directions.
- All subsequent propagation steps take the LPV from the previous iteration as input and propagate as in the first iteration.

The main difference from SHDOM methods is the propagation scheme. Instead of transferring energy from a source cell to its 26 neighbor cells in a regular grid, we propagate to its 6 neighbors only, thus reducing the memory footprint. To preserve as much directional information as possible, we compute the transfer to the faces of these neighbor cells and reproject the intensities to the cells' center (see Figure 5.8). This mimics, but is of course not identical to, the use of 30 unique propagation directions. Please see [Kaplanyan and Dachsbacher 10] for the details. There are two ways to implement this propagation process: scattering and gathering light. The gathering scheme is more efficient in this case due to its cache-friendliness.

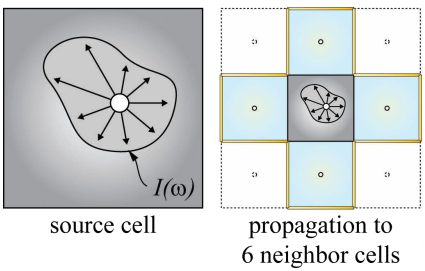


Figure 5.8. Propagation from one source cell (center) to its neighbor cells. Note that we compute the light propagation according to the intensity distribution $I(\omega)$ to all yellow-tagged faces of the destination cells (blueish).

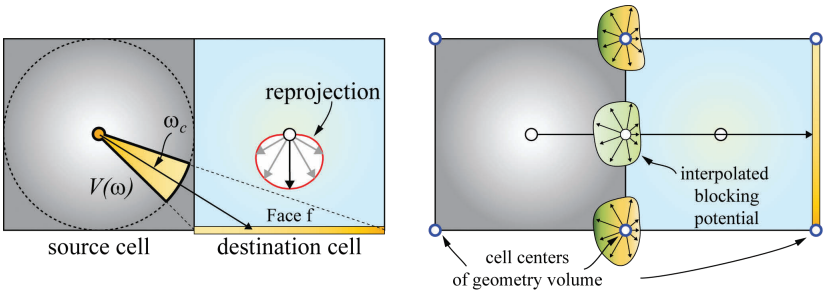


Figure 5.9. Left: Light propagation from the source cell (gray) to the bottom face of the destination cell (blueish). Right: during propagation we can account for occlusion by looking up the blocking potential from the geometry volume.

When propagating the light from a source cell to one face of the destination cell, we compute the incoming flux onto the face using the solid angle $\Delta\omega$ of the face and the central direction ω_c of the propagation cone (see Figure 5.9). The flux reaching the face is then computed as

$$\Phi_f = \frac{\Delta\omega}{4\pi} I(\omega_c),$$

where $I(\omega_c)$ is the intensity of the source cell towards the center of face obtained by evaluating the SH approximation. Here we assume that the intensity does not vary over the solid angle of the face.

Reprojection. The flux incident on a face is then reprojected back into the intensity distribution of the destination cell. The reprojection is accomplished by creating a new virtual surfel at the destination cell's center, pointing toward the face and emitting exactly as much flux as the face received from the propagation (Φ_f):

$$\Phi_l = \int_\Omega \Phi_f \langle n_l, \omega \rangle d\omega = \frac{\Phi_f}{\pi}.$$

Similar to the light injection stage, the corresponding clamped cosine lobe is scaled by Φ_l and accumulated into SH coefficients of the destination cell. In other words, we compute the incoming flux for each face of the destination cell and transform it back into an intensity distribution.

Blocking for indirect shadows. Indirect shadows, i.e., the blocking of indirect light due to scene geometry, can also be incorporated into the LPVs. In order to add indirect shadows, we construct a volumetric representation of the scene's surfaces (see Section 5.4.2). This so-called geometry volume (GV) is a grid of the same resolution as the LPV and stores the blocking potential (also represented as SH)

for every grid cell. The GV is displaced by half the grid size with respect to the LPV. That is, the cell centers of the GV reside on the corners of the LPV cells. Whenever we propagate from a source to a destination cell, we obtain the bilinearly interpolated SH coefficients—at the center of the face through which we propagate—from the GV, and evaluate the blocking potential for the propagation direction to attenuate the intensity. Note that this occlusion should not be considered for the very first propagation step after injection, in order to prevent immediate self-shadowing.

Iterations. The sum of all intermediate results is the final light distribution in the scene. Thus we accumulate the results of every propagation in the LPV into a separate 3D grid. The number of required iterations depends on the resolution of the volume. We recommend using two times the longest dimension of the grid (when not using a cascaded approach). For example, if the volume has dimensions of $32 \times 32 \times 8$, then the light can travel the whole volume in 64 iterations in the worst case (which is a diagonal of the volume). However, when using a cascaded approach it is typically sufficient to use a significantly lower number of iterations.

Illustrative example of light propagation process in the Cornell Box–like scene. The light propagation is shown in Figure 5.10. The top-left image shows the coarse LPV initialized from the RSM in the injection stage. The noticeable band of reflected blue and red colors has a width of one cell of LPV. Note that after four iterations the indirect light is propagated and touches the small white cube. After eight iterations the indirect light has reached to the opposite wall.

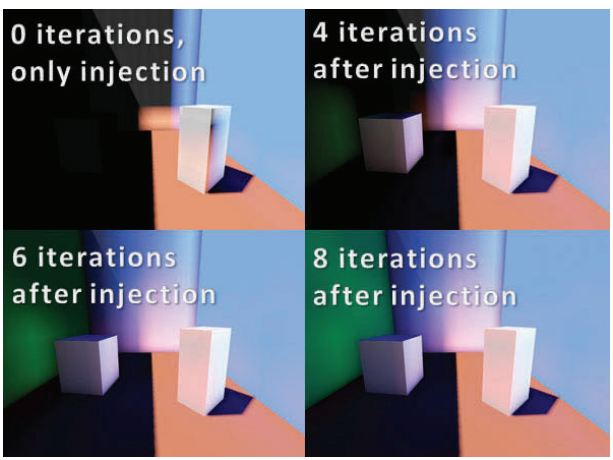


Figure 5.10. Example of the light propagation process in a simple scene.

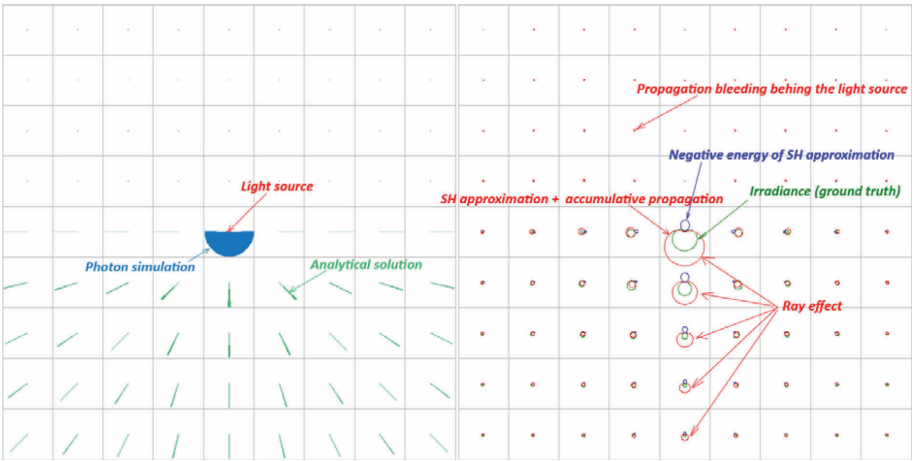


Figure 5.11. Light from a small area light source (smaller than the cell size). The analytical result shows the ground truth solution (left). The light propagation illustrates the ray effect and an unwanted propagation behind the light source (right).

Limitations. The iterative propagation has three main limitations. First, the coarse resolution of the grid does not capture fine details and might even cause light bleeding. Second, due to the SH representation of the intensity distribution and the propagation process itself, the light strongly diffuses and a strictly directed light propagation is not possible. Consequently, there is also no reasonable chance of handling glossy surfaces during propagation. Lastly, the propagation together with the reprojection introduces spatial and directional discretization; this is called the ray effect and is common to all lattice-based methods (see Figure 5.11). Note that some of these artifacts can be suppressed to an acceptable level using cascaded grids to provide finer grids closer to the camera and smart filtering when looking up the LPV.

5.4.2 Indirect Occlusion (Optional Step for Indirect Shadows)

The geometry volume storing the blocking potential of the scene geometry is only required when computing indirect shadows. To this end, we again sample the surfaces using RSMs (and deferred shading buffers, see below) and each texel represents a small part of a surface. Using this sampled scene information, we model the occlusion in the spirit of [Sillion 95], using the accumulated blocking potential of surfels in a grid cell as a probability for blocking light from a certain direction. In this way we can render soft shadows, but shadows of objects smaller than the GV's cell size can of course not be resolved.

Constructing the geometry volume. The blocking potential of a surfel is approximated using the first two bands of the SH basis again. The amount of blocking by one surfel depends on its size and the cosine of the angle between its normal and the light direction in question. The blocking probability of a single surfel with area A_s (computed based on the RSM's texel size in world space), and normal n_s in a cell of grid size s is

$$B(\omega) = \frac{A_s \langle n_s | \omega \rangle}{s^2}.$$

Note that we assume that scene objects are closed surfaces. This is important and allows us to use a clamped cosine lobe for the blocking potential again, as low-order SH projections of absolute cosine lobes tend to degrade to near isotropic functions. Similar to the VPL injection, we accumulate the SH projections of the blocking potential into the GV.

Reusing G-buffers from RSMs and cameras. Aiming at fully dynamic scenes without precomputation requires the creation of the GV—and thus the surface sampling—on the fly. First of all, we can reuse the sampling of the scene's surfaces that is stored in the depth and normal buffers of the camera view (when using a deferred renderer), and in the RSMs that have been created for the light sources. RSMs are typically created for numerous light sources and thus already represent a dense sampling of large portions of the scene. It is, at any time, possible to gather more information about the scene geometry by using depth-peeling for the RSMs or the camera view.

The injection (i.e., determining the blocking potential of a GV's cell) has to be done using separate GVs for every RSM or G-buffer in order to make sure that the same occluder (surfel) is not injected multiple times from different inputs. Afterwards, we combine all intermediate GVs into a single final GV. We experimented using the maximum operator for SH coefficients. Although this is not correct, it yields plausible result when using 2 SH-bands only and clamping the evaluation to zero, and thus yields more complete information about light-blocking surfaces.

5.4.3 Final Scene Illumination

The accumulated results of the propagation steps represent the light distribution in the scene. In the simplest form we query the intensity by a trilinearly interpolated lookup of the SH coefficients and compute the reflected radiance of a surface. We then evaluate the SH-intensity function for the negative normal of the surface, similar to irradiance volumes [Greger et al. 97], and next convert it into incident radiance. For this we assume that the cell's center (where the intensity is assumed to reside) is half the grid size s away from the surface.

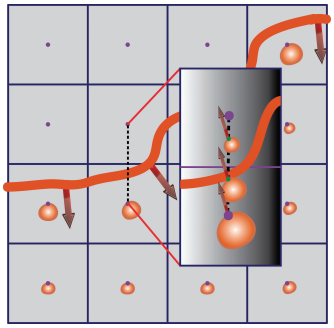


Figure 5.12. The interpolation of the intensity stored in the grid induced by the orange surface between two cells. Note the wrong result of linear interpolation behind the emitting object itself. Also note that the gradient is opposing in this case.

LPVs and light prepass rendering. In light prepass architecture [Engel 09] of the CryENGINE 3, the LPV lighting is directly rendered into the diffuse-light-accumulation buffer on top of multiple ambient passes. This allows us to use optimizations, such as stencil prepass and depth-bound tests, for this pass as well. Moreover, it is also possible to compose complex layered lighting.

Improving filtering by using directional derivatives. The trilinear interpolation of SH coefficients for looking up the LPV can cause serious artifacts such as self-illumination and light bleeding. We found that a damping factor based on the directional derivative of the intensity distribution greatly reduces these artifacts. For a surface location x with normal n, we determine the trilinearly interpolated SH coefficients c and the directional derivative in the normal direction ∇c (computed via final differencing) (see Figure 5.12):

$$\nabla c(x) = \frac{c(x) - c(x+n)}{\|n\|} = c(x) - c(x+n).$$

Whenever the derivative is large, and c and ∇c are deviating, we damp c before computing the lighting.

This additional filtering during the final rendering phase yields sharper edges between lit and shadowed areas.

5.4.4 A Multi-Resolution Approach using Cascaded Light Propagation Volumes

So far we have considered a single LPV only; however, a regular 3D grid does not provide enough resolution everywhere in scenes with large extent, or otherwise requires prohibitively high memory storage and propagation cost. In this section

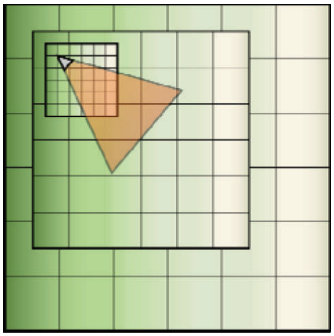


Figure 5.13. Cascaded approach. Cascades are nested and attached to the camera; note that cascades are slightly shifted toward the view direction.

we propose a multi-resolution approach to this problem keeping both memory consumption and computation cost low, even for large scenes (see Figure 5.13).

Cascaded LPVs. In spirit of the cascaded shadow map (CSM) approach [Engel 05, Dimitrov 07], we use multiple nested grids, or cascades, for light propagation that move with the camera. For every cascade, we not only store an LPV, but we also create a separate RSM for every light source, where the RSM resolution is chosen proportional to the grid cell sizes as described above. However, unlike CSMs, the indirect lighting can also be propagated from behind to surfaces in front of the camera (see Figure 5.14). In practice we use a 20/80 ratio for cascade shifting, i.e., the cascades are not centered around the camera but shifted in view directions such that 20% of their extent is located behind the camera. Usually it is sufficient to use three nested grids with a respective double size.

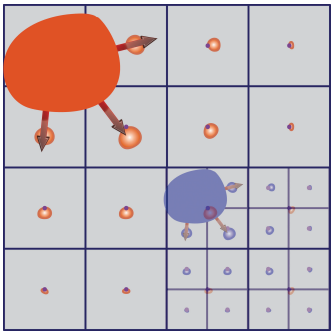


Figure 5.14. Indirect light propagated from objects of different sizes in the respective resolution.

The idea of LPVs naturally transfers to the multi-resolution approach, and every cascade requires RSM generation, injection, propagation, and LPV lookup almost as before. However, before the RSM generation starts, we determine for every object in the scene to which cascade it belongs, and thus in which RSM it has to be considered. This ensures that we do not account for the indirect light contribution of an object multiple times.

Light propagation with cascades. So far we detailed light propagation within a single grid, but handling multiple cascades simultaneously imposes new questions: How do we propagate across different cascades? How do we combine the contributions of different cascades for the final rendering? In this section we propose two options depending on whether indirect shadows are a required feature or not.

If no indirect shadows are required, we can handle the cascades independent from each other, and the multi-resolution approach is straightforward to implement. With indirect shadows, we have to correctly deal with light propagating across the edges of cascades and, of course, blocking.

Cascaded LPVs without indirect shadows. Assuming that light propagates without blocking, we can completely decouple the cascades and compute a LPV solution with the following steps:

- Every RSM for a each cascade should contain unique objects causing indirect light. Objects are normally rendered into the RSM for the cascade for which they have been selected; in RSMs for other cascades they are rendered with black albedo in order to prevent indirect light contribution, but correctly rendering direct shadows.
- The propagation is done for each cascade independently.
- The scene is illuminated by the accumulated contribution of all cascades.

In this case, we determine the respective cascade for every object by estimating its contribution to the indirect illumination, which in turn heavily depends on its surface area and the distance to the camera. To this end, we account for large and distant objects in the coarser cascades while injecting smaller, close objects into the finer grids. Note that this means that distant, small objects might not be considered during the computation (if they lie outside the finer cascades). However, the indirect illumination stemming from such objects typically has a significant contribution only within a certain (small) proximity of the object.

Cascades with indirect shadows. When accounting for light blocking we cannot decouple the propagation process of the cascades. In this case every object is injected into the finest grid at its respective location. This also means that those parts of coarser grids that are overlapped by finer grids are empty and not used during the propagation. Although we apply the propagation steps to each

cascade separately, we have to make sure that light leaving one grid can further propagate in the next grid. This is simple when the grid cell sizes of two cascades differ by a power-of-two factor. Note that intensity has to be redistributed when propagating light from a single cell of a coarse grid to multiple cells of a fine grid, and accumulated when propagating light from multiple fine grid cells to a coarse grid cell (as intensity already accounts for the surface area).

Stable lighting through snapping. The LPV cascades are oriented along the world space axes and do not rotate, only translate, with the camera. It is important to snap the cascade positions in world-space to multiples of the cell sizes. This ensures that the distribution of surfels in the injection stage is stable and independent of the translation of the LPVs.

5.5 Optimizations

5.5.1 General Optimizations

The scene illumination stage with a final LPV is usually a very expensive pass (see the timing table in Section 5.6.1). It is important to note that the hardware capability to render into a volume texture tremendously improves performance at this stage. This simplifies the shader workload as the emulation of a trilinear filtering in the pixel shader is not necessary. Instead, the hardware unit is utilized and cache coherency is highly improved due to the optimized memory layout of a swizzled 3D texture.

Unfortunately, not every platform supports rendering into a 3D texture. However, it is possible to inject the VPLs directly into the volume texture on consoles. To do so, the 3D texture should be treated as a horizontally unwrapped 2D render target; note that this is not possible with the Microsoft DirectX 9.0 API.

Using an 8 bit/channel texture format for the LPV has proven to be sufficient for diffuse indirect lighting stored in two bands of SH. Note that this detail is very important because it speeds up the final rendering pass significantly due to decreased bandwidth and texture-cache misses.

5.5.2 Temporal Coherence

As the diffuse indirect lighting is inherently smooth, the observer is usually quite unaware of temporally changing smooth gradients. Exploiting this fact, we recommend caching each resulting LPV and reusing the results across several frames. The temporal reprojection can easily be done in a straightforward way with accumulation of the old results and a smooth substitution by the newly propagated results.

In practice, we perform the full pass of LPV computation (RSM generation, injection, and propagation) each fifth frame. With the target frame rate of 30 frames/second that means that the refresh rate for the indirect lighting is six

times/second. This proved to be sufficient in the vast majority of cases. The workload can be distributed either stage-by-stage across several frames or can be executed all at once (RSM rendering, injection, and propagation) during one frame. In CryENGINE 3 we do not spread separate stages; however, we update different cascades in different frames. In addition, we use different update frequencies for different cascades of cascaded shadow maps. The LPV update workload is interleaved with updates of shadow map cascades, which ensures that we have a balanced rendering and consistent performance.

5.5.3 Xbox 360-Specific Optimizations

When doing the injection directly into a 3D texture on the Xbox 360, we recommend redundantly unwrapping this texture vertically as well (so a $32 \times 32 \times 32$ 3D texture becomes a 32×1024 2D texture after being aliased as a 2D render target, but it should additionally be duplicated vertically). This trick significantly reduces the h/w bank conflicts caused by multiple surfels being written to the same pixel.

Note that the redundant copies of the texture should be accumulated together afterwards and added into the final 3D texture. Also, the resolve GPU command from the EDRAM into a 3D texture has some API issues; a work-around solution for these issues is proposed in the Appendix of [Kaplanyan 09].

5.5.4 PlayStation 3-Specific Optimizations

The 3D texture is essentially implemented as an array of contiguous 2D textures. This means that the layout of unwrapped 2D render targets should be vertical on PlayStation 3 in order to match the memory layout of the 3D texture. Then the injection goes directly into a 3D texture and can be done by using the simple memory aliasing of this texture as a 2D render target. The RSX pipeline should obviously be flushed after the injection rendering in order to make sure that all the surfels are injected before the propagation is started.

Another suggestion for speeding up the final illumination pass is to alias the target buffer as a $2\times$ MSAA and do the shading in a half-resolution horizontally. This is possible only on PlayStation 3 because the layout of samples in the sampling scheme matches the layout of the render target with the same pitch. This trick can help to reduce the pixel work without noticeable quality degradation, as the diffuse indirect lighting typically exhibits low frequencies only.

5.6 Results

In this section we provide several example screenshots showing diffuse global illumination in Crysis 2 (see Figure 5.15). In Crysis 2 we use one to three cascades depending on the graphics settings, platform, and level specifics.

Figure 5.15. In *Crysis 2* we use from one to three cascades depending on graphics settings, platform, and level specifics.

5.6.1 Tools and Tweaking Parameters for Artists

Because artists want to have more control over the indirect lighting, it was decided to expose multiple controlling parameters for the technique:

- *Global intensity of indirect lighting.* This parameter helps to tweak the overall composition and can also be tweaked during the game play from the game logics.
- *Per-object indirect color.* This parameter affects only the color and the intensity of indirectly bounced lighting from a specific object. It is mostly used to amplify or attenuate the contribution of some particular objects into indirect lighting.
- *Propagation attenuation.* This parameter is used to tweak the attenuation of the indirect color.
- *Per-object "indirect receiver" parameter for particles.* This is mostly used to tag some particle effects as those illuminated by the indirect lighting (disabled by default).

These parameters are frequently used by artists to tweak some particular places and moments in the game. They proved to be particularly important for cut scenes and some in-game movies.

5.6.2 Timings

Detailed timings for a single cascade for the Crytek Sponza[2] scene are provided in Table 5.1. Note that the timings for multiple cascades can be estimated by multiplying the timings for a single cascade by the number of cascades when using the cascaded approach, as the work is spread across several RSMs.

For all screenshots in this chapter we used the same settings: the size of the LPV grid is $32 \times 32 \times 32$, the propagation uses 12 iterations and 1 1 cascade, and the rendering was at 1280×720 resolution (no MSAA). The cost of the final illumination pass obviously depends on the target buffer resolution. The RSM size is 256^2 for the timings with the NVIDIA GTX285 and 128^2 for both consoles. Of course the cost of the RSM rendering also depends on the scene complexity.

Stage	**NVIDIA GTX 285**	**Xbox**	**PlayStation**
RSM Rendering	0.16 (256^2)	0.5 (128^2)	0.8 (128^2)
VPL Injection	0.05	0.2	0.4
Propagation (12 iterations)	0.8	0.8	0.7
Final illumination (720p)	2.4 (RGBA16F format)	2.0	1.5
Total (per frame)	**1.5**	**1.1**	**1.2**

Table 5.1. Detailed timings for a single cascade for the Crytek Sponza scene. All measurements are in milliseconds for the individual stages.

5.7 Conclusion

In this chapter we described a highly parallel production-ready and battle-tested (diffuse) global illumination method for real-time applications without any precomputation. To our knowledge, it is the first technique that employs a light propagation for diffuse global illumination. It features a very consistent and scalable performance, which is crucial for real-time applications such as games. We also described how to achieve indirect illumination in large-scale scenes using a multi-resolution light propagation.

We demonstrated our method in various scenes in combination with comprehensive real-time rendering techniques. In the future we would like to reduce the limitations of our method by investigating other grid structures and other adaptive schemes, where the compute shaders of DirectX 11 will probably of great help.

[2]The Crytek Sponza scene is the original Sponza Atrium scene improved for global illumination experiments. This scene is granted to the rendering community and can be downloaded from this link: http://crytek.com/cryengine/cryengine3/downloads.

5.8 Acknowledgments

Thanks to Sarah Tariq and Miguel Sainz for implementing this technique as a sample of the NVIDIA SDK[3]. And of course the whole Crytek R&D team as well as all others who helped and discussed the real-time graphics with us!

Bibliography

[Chandrasekhar 50] S. Chandrasekhar. *Radiative Transfer*. New York: Dover, 1950.

[Dachsbacher and Stamminger 05] C. Dachsbacher and M. Stamminger. "Reflective Shadow Maps." In *Proc. of the Symposium on Interactive 3D Graphics and Games*, pp. 203–213. Washington, DC: IEEE Computer Society, 2005.

[Dimitrov 07] R. Dimitrov. "Cascaded Shadow Maps." Technical report, NVIDIA Corporation, 2007.

[Engel 05] W. Engel. "Cascaded Shadow Maps." In *Shader X^5*, pp. 129–205. Hingham, MA: Charles River Media, 2005.

[Engel 09] W. Engel. "Light Prepass." In *Advances in Real-Time Rendering in 3D Graphics and Games Course—SIGGRAPH 2009*. New York: ACM Press, 2009.

[Evans 98] K. F. Evans. "The Spherical Harmonic Discrete Ordinate Method for Three-Dimensional Atmospheric Radiative Transfer." In *Journal of Atmospheric Sciences*, pp. 429–446, 1998.

[Greger et al. 97] G. Greger, P. Shirley, P. Hubbard, and D. Greenberg. "The Irradiance Volume." *IEEE Computer Graphics & Applications* 18 (1997), 32–43.

[Kaplanyan and Dachsbacher 10] Anton Kaplanyan and Carsten Dachsbacher. "Cascaded Light Propagation Volumes for Real-Time Indirect Illumination." In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 99–107, 2010.

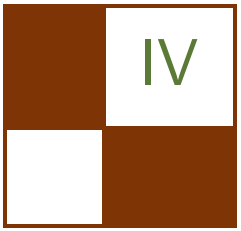
[Kaplanyan 09] A. Kaplanyan. "Light Propagation Volumes in CryEngine 3." In *Advances in Real-Time Rendering in 3D Graphics and Games Course—SIGGRAPH 2009*. New York: ACM Press, 2009.

[Sillion 95] F. Sillion. "A Unified Hierarchical Algorithm for Global Illumination with Scattering Volumes and Object Clusters." *IEEE Trans. on Visualization and Computer Graphics* 1:3 (1995), 240–254.

[Sloan 08] P.-P. Sloan. "Stupid Spherical Harmonics (SH) Tricks." In *GDC'08*, 2008.

[Tatarchuk 04] N. Tatarchuk. "Irradiance Volumes for Games." Technical report, ATI Research, Inc., 2004.

[3]Publicly available for download at: http://developer.nvidia.com/object/sdk_home



IV Shadows

In Part IV we will cover various algorithms that are used to generate shadow data. Shadows are the dark companions of lights and although both can exist on their own, they shouldn't exist without each other in games. Achieving good visual results in rendering shadows is still considered one of the particularly difficult tasks of graphics programmers. One of the trends in this shadow section is the description of implementations that achieve perceptually correct looking shadows; in real-time graphics terminology, called soft shadows. Three articles build on Randy Fernando's work, "Percentage-Closer Soft Shadows," that is now—five years after it was published—still an efficient way to implement soft shadows on the current generation of hardware.

The first article in this section, "Variance Shadow Maps Light-Bleeding Reduction Tricks," by Wojciech Sterna, covers techniques to reduce light bleeding. There is also an example application that shows the technique.

Pavlo Turchyn covers fast soft shadows with adaptive shadow maps—as used in *Age of Conan*—in his article "Fast Soft Shadows via Adaptive Shadow Maps." The article describes the extension of percentage-closer filtering to adaptive shadow maps that was implemented in the game. Turchyn proposes a multiresolution filtering method in which three additional, smaller shadow maps with sizes of 1024×1024, 512×512 and 256×256 are created from a 2048×2048 shadow map. The key observation is that the result of the PCF kernel over a 3×3 area of a 1024×1024 shadow map is a reasonably accurate approximation for filtering over a 6×6 area of a 2048×2048 shadow map. Similarly, a 3×3 filter kernel of a 256×256 shadow map approximates a 24×24 area of a 2048×2048 shadow map.

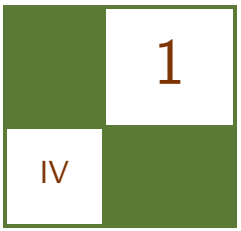
The article "Adaptive Volumetric Shadow Maps" by Marco Salvi et al. describes a new approach for real-time shadows that supports high-quality shadowing from dynamic volumetric media such as hair, smoke, and fog. Adaptive volumetric shadow maps (AVSM) encode the fraction of visible light from the light source over the interval $[0, 1]$ as a function of depth at each texel. This transmittance function and the depth value are then stored for each texel and sorted front-to-back. This is called the AVSM representation. This AVSM representation is generated by first rendering all visible transparent fragments in a linked list (see the article "Order-Independent Transparency Using Per-Pixel

Linked Lists in DirectX 11" for the description of per-pixel linked lists). In a subsequent pass, those linked lists are compressed into the AVSM representation, consisting of the transmittance value and the depth value.

Another article that describes the fast generation of soft shadows is "Fast Soft Shadows With Temporal Coherence" by Daniel Scherzer et al. The light source is sampled over multiple frames instead of a single frame, creating only a single shadow map with each frame. The individual shadow test results are then stored in a screen-space shadow buffer. This buffer is recreated in each frame using the shadow buffer from the previous frame as input. This previous frame holds only shadowing information for pixels that were visible in the previous frame. Pixels that become newly visible in this frame due to camera or object movement have no shadowing information stored in this buffer. For these pixels the article describes a spatial-filtering method to estimate the soft shadow results. In other words the main idea of the algorithm described in the article is to formulate light-source area sampling in an iterative manner, evaluating only a single shadow map per frame.

The last article in the section, "MipMapped Screen Space Soft Shadows," by Alberto Aguado et al. uses similar ideas as the other two soft-shadow articles. Soft shadows are generated with the help of mipmaps to represent multi-frequency shadows for screen-space filtering. The mipmap has two channels; the first channel stores the shadow-intensity values and the second channel stores screen-space penumbra widths. Shadow values are obtained by filtering while penumbrae widths are propagated by flood filling. After the mipmap is generated, the penumbrae values are used as indices to the mipmap levels. Thus, we transform the problem of shadow generation into the problem of selecting levels in a mipmap. This approach is extended by including layered shadow maps to improve shadows with multiple occlusions.

—Wolfgang Engel



Variance Shadow Maps Light-Bleeding Reduction Tricks

Wojciech Sterna

1.1 Introduction

Variance Shadow Maps (VSMs) were first introduced in [Donnelly and Lauritzen 06] as an alternative to bilinear percentage closer filtering (PCF) to speed up rendering of smoothed shadows. The algorithm is relatively inexpensive, easy to implement, and very effective in rendering shadows with large penumbra regions. However, VSM has one major drawback—apparent light-bleeding—which occurs when two or more shadow casters cover each other in light-space. This article will show techniques that help to reduce the light-bleeding artifacts in VSM.

1.2 VSM Overview

The idea of variance shadow mapping is to store in the shadow map, instead of a single depth value, a distribution of depth values over some region and to use elementary statistics to evaluate the shadowing term. This approach makes it possible to use filtering methods (bilinear, trilinear, blurring) on the shadow map. A common option is to use Gaussian blur on the shadow map to achieve soft shadows in $O(n)$ time. This is a great advantage over traditional PCF which requires $O(n^2)$ time to achieve the same effect.

To generate the variance shadow map, two values must be written into it. The first is simply a distance from a light source to a pixel, as with traditional shadow mapping (one thing that is important here is that this distance should have a linear metric). The second component of the shadow map is a square of the first component.

Once the shadow map has been prepared (it contains both depth and a square of depth), additional filtering can be applied to it. To achieve good-looking soft shadows, a separable Gaussian filter with 5×5 taps can be used.

To estimate the shadow contribution from such a defined VSM, so-called Chebyshev's inequality (one-tailed version) can be used to estimate the shadowing term:

$$P(O \geq R) \leq p_{\max}(R) \equiv \frac{\sigma^2}{\sigma^2 + (R - \mu)^2}, \quad \text{where} \quad \mu < R \tag{1.1}$$

The variable O is an occluder's depth (shadow map's texel), R is a receiver's depth (distance from a pixel being shaded to a light source), σ^2 is the variance and μ is the mean. The term $P(O \geq R)$ can roughly be interpreted as a probability of a point (at distance R) being lit (unshadowed by a point at distance O), which is the exact value we wish to know, and Chebyshev's inequality gives us an upper-bound to this value.

The mean and variance in Equation (1.1) are computed from the first and second statistical moments:

$$\begin{aligned} M_1 &= E(O) \\ M_2 &= E(O^2) \\ \mu &= M_1 = E(O) \\ \sigma^2 &= M_2 - {M_1}^2 = E(O^2) - E(O)^2 \end{aligned}$$

In fact, the first moment is what is actually stored in the first channel of the shadow map, and the second moment in the second channel of the shadow map. That's why the shadow map can be additionally prefiltered before its use—the moments are defined by the expectation operator which is linear and can thus be linearly filtered.

A sample implementation of standard VSM is shown in Listing 1.1.

```
float VSM(float2 projTexCoord, float depth, float bias)
{
    float2 moments = tex2D(shadowMap_linear, projTexCoord).rg;

    if (moments.x >= depth - bias)
        return 1.0f;

    float variance = moments.y - moments.x*moments.x;
    float delta = depth - moments.x;
    float p_max = variance / (variance + delta*delta);

    return saturate(p_max);
}
```

Listing 1.1. Standard VSM implementation.

Note that the given Chebyshev's inequality (its one-tailed version) is undefined for cases in which $\mu \geq R$. Of course, in such a case a point is fully lit, so the function returns 1.0; otherwise, $p_{\max}$ is returned.

1.3 Light-Bleeding

Light-bleeding (see Figure 1.1) occurs when two or more shadow casters cover each other in light-space, causing light (these are actually soft edges of shadows of the objects closest to the light) to bleed onto the shadows of further (from the light) objects. Figure 1.2 shows this in detail.

As can be seen from the picture in Figure 1.2, object C is lit over a filter region. The problem is that when estimating the shadow contribution for pixels of object C over this region, the variance and mean that are used are actually based on the samples from object A (red line) and visible samples from object B (green line) (the shadow map consists of pixels colored by red and green lines), whereas they should be based on samples from object B only (green and blue lines). Moreover, the greater the ratio of distances $\frac{\Delta x}{\Delta y}$ (see Figure 1.2), the more apparent the light-bleeding is on object C.

The VSM is not capable of storing more information (about occluded pixels of object B for instance). This is not even desirable since we want to keep the

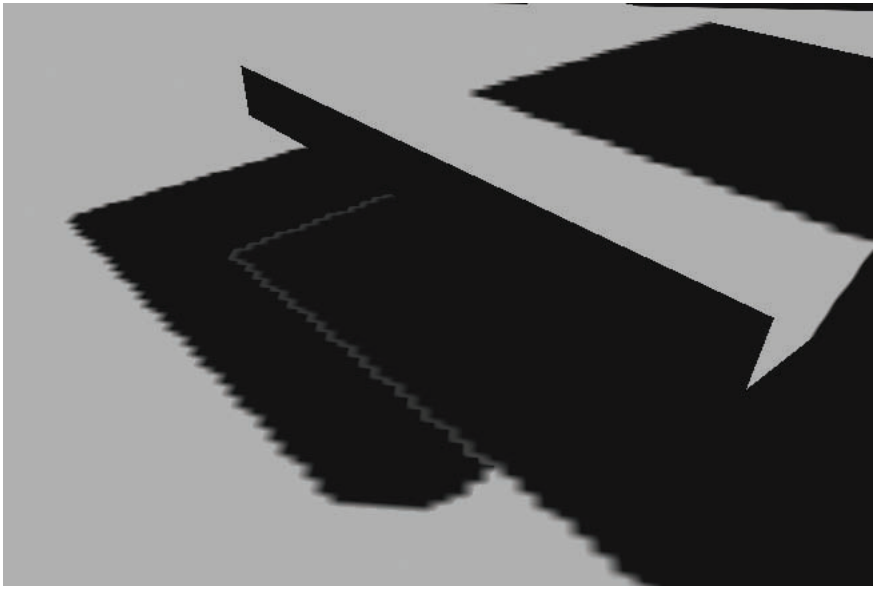


Figure 1.1. Light-bleeding.

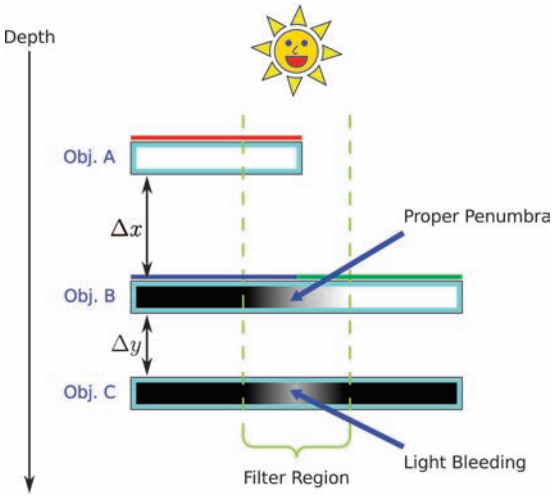


Figure 1.2. How light-bleeding occurs.

algorithm simple and don't want to raise its memory requirements. Fortunately, the following sections in this chapter will present a few very simple tricks that can greatly reduce the problem.

1.4 Solutions to the Problem

1.4.1 Cutting the Tail Off

The first way of reducing light-bleeding was simply cutting off the tail of the $p_{\max}$ function. Put simply, it's about subtracting some fixed value from the result of $p_{\max}$. This way, incorrect penumbra regions will get darker but so will the correct ones. However, intensities of incorrect penumbra regions never reach 1 (as shown in [Lauritzen 07]) so the final result can still look good when the cutting value is chosen wisely.

Figure 1.3 shows a side-by-side comparison of standard VSM implementation and the same VSM but with $p_{\max}$ cut off by a value of 0.15. Although the light-bleeding is still present it's much less obvious.

As we mentioned earlier, the light-bleeding gets more noticeable when the ratio $\frac{\Delta x}{\Delta y}$ is big. In such a scenario (high depth complexity) it is next to impossible to eliminate the problem by applying only a cutting value. More sophisticated methods are necessary.

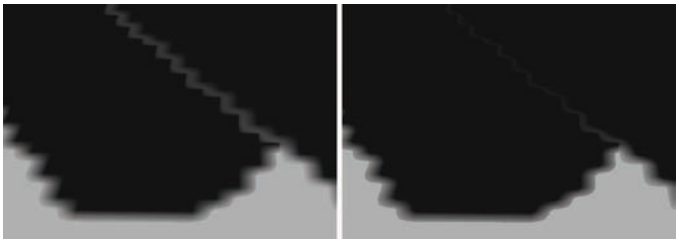


Figure 1.3. VSM and VSM with $p_{\max}$ cut off by 0.15.

1.4.2 Applying VSM to Shadow Boundaries

One of the possibilities for reducing the problem of light-bleeding is, simply to avoid it. The trick is to combine standard depth-comparison shadow mapping with VSM applied only to regions that really need it—shadow boundaries.

The problem with this idea is that the greater the penumbra region, the harder it gets to properly detect shadow boundaries, since more samples are needed. However, if one decides not to blur the shadow map and to rely only on standard hardware linear (or trilinear) filtering, four samples are enough to detect shadow boundaries; light-bleeding free VSM is achieved with the resulting performance comparable to that from standard VSM.

Figure 1.4 shows a side-by-side comparison of standard VSM and the combination of classic shadow mapping with VSM applied only to shadow boundaries.

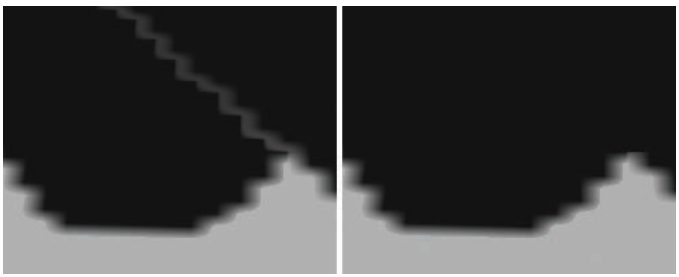


Figure 1.4. VSM and VSM applied to shadow boundaries.

1.4.3 Exponential Variance Shadow Maps

EVSM is a combination of two techniques—variance shadow maps and exponential shadow maps, which were described in [Salvi 08]. The idea was first presented in [Lauritzen 08] and it is surprising that it is not recognized among developers, for it is able to almost completely eliminate the light-bleeding.

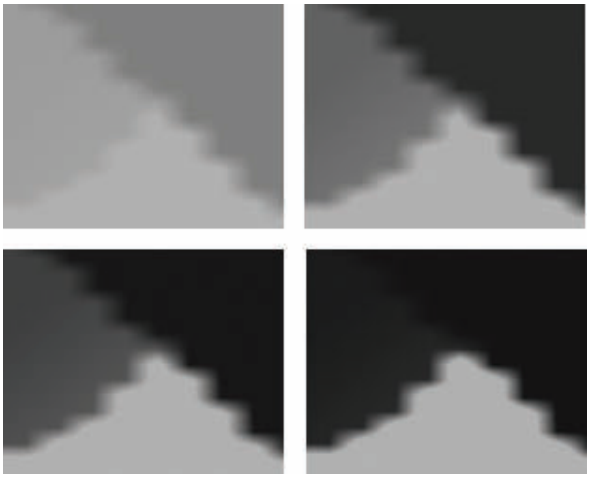


Figure 1.5. ESM with the following constants: 10, 50, 90, 200.

ESMs work similarly to VSMs. A very nice feature of ESM is that it requires only a single-channel shadow map, which stores the first moment. The algorithm also uses statistical methods to estimate the shadowing term and it does so by using the so-called Markov inequality (as opposed to VSM which uses Chebyshev's inequality):

$$P(O \geq R) \leq \frac{E(O)}{R} \tag{1.2}$$

Using Markov inequality as given in Equation (1.2) doesn't provide a good shadow approximation; shadows suffer from a sort of global light-bleeding. However, [Salvi 08] shows that it can be transformed to the following representation:

$$P(e^{kO} \geq e^{kR}) \leq \frac{E(e^{kO})}{e^{kR}} \tag{1.3}$$

Constant k determines how good the approximation is—the greater the value, the better the approximation. Unfortunately, large values cause precision loss, and shadow boundaries become sharper, so a compromise must be found. Figure 1.5 shows a comparison of ESM with different values of constant k.

ESM is simpler, faster, and has a tweakable light-bleeding parameter that makes it more practical than VSM in many cases. However, a very promising idea is the combination of these two techniques—EVSM. Instead of storing depth and a square of depth in the shadow map, we store an exponential of depth and a square of exponential of depth. The exponential function has the effect of decreasing the ratio $\frac{\Delta x}{\Delta y}$ and thus reduces the VSM-like light-bleeding.

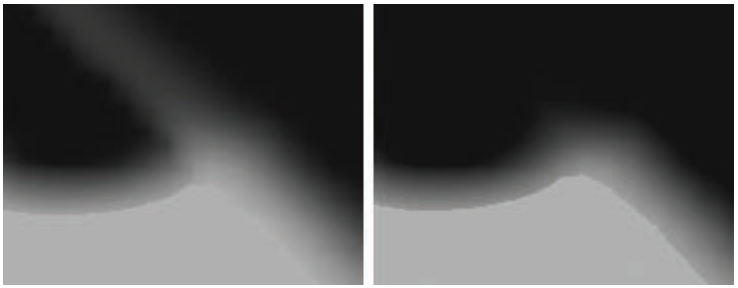


Figure 1.6. VSM and EVSM with p_{max} cut off by 0.05 and $k = 50$.

EVSM suffers from light-bleeding only in cases when both VSM and ESM fail, which rarely happens. A very important feature of EVSM is that the light-bleeding can be controlled by two factors: VSM with the tail cut off and ESM with k constant. Careful adjustment of these two will lead to very pleasing and accurate soft shadows.

Figure 1.6 shows a side-by-side comparison of standard VSM and EVSM.

1.5 Sample Application

In the web materials accompanying this book there is a demo presenting all of the techniques described in this chapter.

Here is the key configuration:

- WSAD + mouse—camera movement;
- Shift—speeding up;
- F1 - F5—change of shadowing technique;
- E/Q—turn on/off shadow map Gaussian blurring;
- R/F—increase/decrease VSM tail cut off parameter;
- T/G—increase/decrease ESM constant k.

The core of the demo are two shader files: `shadow_map.ps` and `light.ps`. Implementations of all algorithms described here can be found in these files.

1.6 Conclusion

Variance shadow mapping has already proven to be a great way of generating soft shadows. The algorithm is easy to implement, fast, and utilizes hardware features of modern GPUs. Despite its advantages, VSM also introduces some

problems. The worst one is light-bleeding, which was the subject of discussion in this chapter.

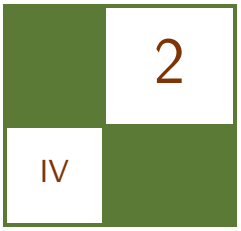
Bibliography

[Donnelly and Lauritzen 06] William Donnelly and Andrew Lauritzen. "Variance Shadow Maps.", 2006. Available online (http://www.punkuser.net/vsm/).

[Lauritzen 07] Andrew Lauritzen. "Summed-Area Variance Shadow Maps." In *GPU Gems 3*, Chapter II.8. Reading, MA: Addison-Wesley, 2007.

[Lauritzen 08] Andrew Lauritzen. Master's thesis, University of Waterloo, 2008.

[Salvi 08] Marci Salvi. "Rendering Filtered Shadows with Exponential Shadow Maps." In *ShaderX*^{6}, Chapter IV.3. Hingham, MA: Charles River Media, 2008.



Fast Soft Shadows via Adaptive Shadow Maps

Pavlo Turchyn

We describe a *percentage-closer filtering* (PCF) based extension to an *adaptive shadow maps* (ASM) algorithm [Fernando et al. 01]. Our method enables high-quality rendering of large penumbrae with minimal size of filter-kernel size. It also offers better image quality compared to other PCF-based soft shadows algorithms, such as *percentage closer soft shadows* (PCSS).

2.1 Percentage-Closer Filtering with Large Kernels

Filtering is an important part of every shadow mapping implementation because it serves multiple goals: it reduces aliasing artifacts and allows the creation of soft shadows to improve image realism. Aliasing artifacts manifest themselves when there are either too many shadow map texels per screen area (resulting in the same type of noise that can be observed on regular color textures without mipmapping), or there are too few shadow map texels per screen area, so the shadow map cannot plausibly represent the correct shapes of shadows. However, filtering alone can reduce only the visual impact of undersampling to a certain extent.

PCF is a common method of shadow map filtering. Given shadow map coordinates uv, and depth value d, a PCF computes the following weighted sum:

$$\texttt{PCF}(\texttt{DepthTexture}, uv, d, n) = \sum_{i=1}^{n} \big(\texttt{Weight}[i] * (\texttt{tex2D}(\texttt{DepthTexture}, uv + \texttt{Offset}[i])).\mathrm{r} > d\big).$$

The tables `Weight` and `Offset` hold weights and the texture coordinates offsets, respectively. The choice of `Weight` and `Offset` defines performance and quality of rendering. The computationally fastest way is to use constant tables. In more elaborate schemes, the tables are constructed based on coordinates uv.

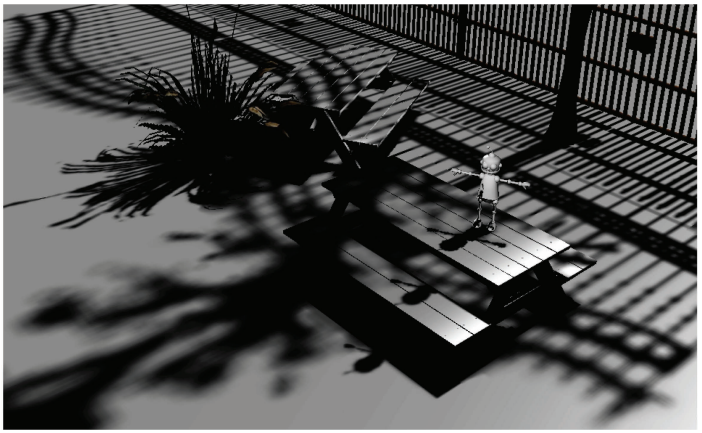


Figure 2.1. Soft shadows in *Age of Conan*.

The number of samples n is chosen depending on the size of the area over which the filtering is performed. Summing over the area of $m \times m$ shadow map texels would require at least $n = m^2$ samples if we want to account for all texels. However, such a quadratic complexity of the filter makes PCF impractical for filtering over large areas. For instance, if the size of a shadow map texel's projection onto a scene's geometry is 0.1 m, and the desired penumbra size is about 1.5 meters, then we need to apply PCF over $(1.5/0.1) \times (1.5/0.1) = 15 \times 15$ area, which in turn gives $n = 225$.

It is possible to use PCF with large kernels in time-sensitive applications by decreasing the number of samples, so that $n \ll m^2$, and distributing the samples pseudorandomly over the summation area. Such an approach produces penumbra with noise artifacts as shown in Figure 2.2. A screen-space low-pass filtering can be used for suppressing the noise, but such a post-filtering removes all high-frequency features within penumbra indiscriminately. Moreover, preventing shadows from bleeding into undesired areas in screen space may require the use of relatively expensive filters,(e.g., a bilateral filter).

We propose a *multiresolution filtering* (MRF) method that attempts to alleviate the PCF problem described above. The idea is as follows: When we create the standard shadow map for the scene, we also create versions of it with progressively lower resolutions. For example, if the shadow map's resolution is 2048×2048, we create three additional shadow maps: 1024×1024, 512×512, and 256×256. The key observation is that the result of PCF over a 3×3 area of a 1024×1024 shadow map is a reasonably accurate approximation for filtering over 6×6 area of 2048×2048 map. Similarly, PCF over a 3×3 area of a 256×256 shadow map approximates a 6×6 filter for a 512×512 map, a 12×12 filter for a 1024×1024 map, and a 24×24 filter for a 2048×2048 map. Thus, in order to

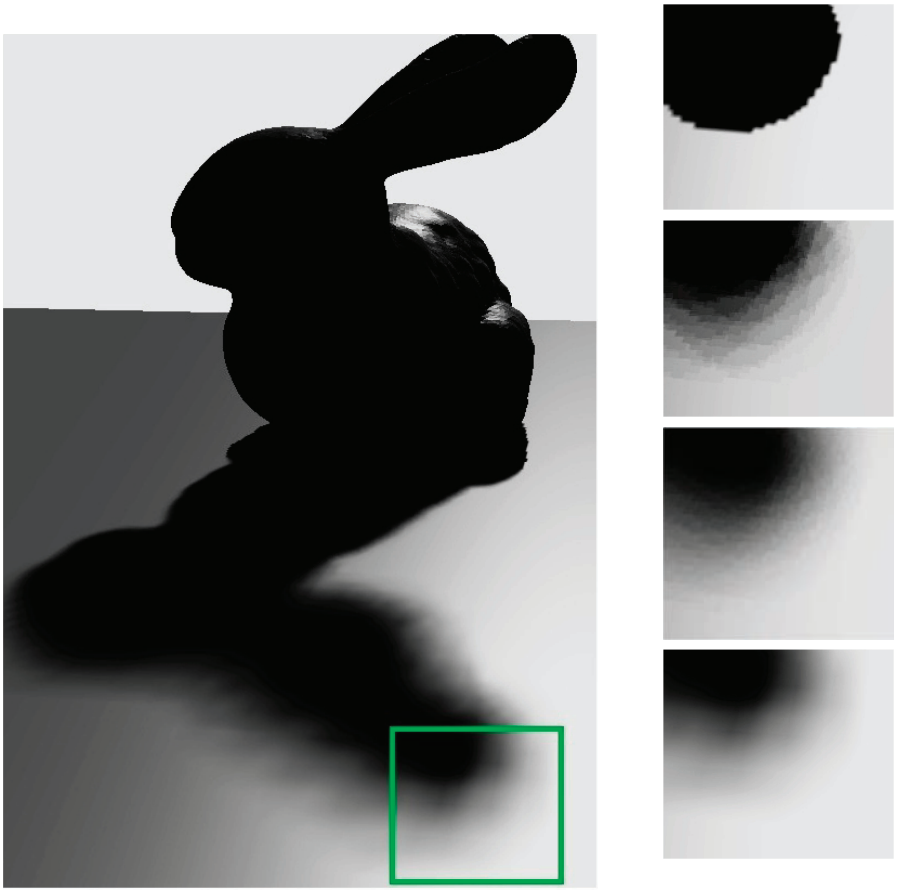


Figure 2.2. Filtering comparison. From top to bottom: bilinear PCF; 24×24 PCF filter with 32 samples, frame rendering time on Radeon 4870 is 3.1 ms; 24×24 PCF filter with 64 samples, 6.7 ms; 24×24 MRF filter (3×3 PCF), 3.1 ms.

approximate PCF with large kernels, we apply PCF with a small kernel size to a shadow map with reduced resolution.

Approximating arbitrary kernel size. Let us number shadow maps starting with zero index assigned to the shadow map with the highest resolution

```
sampler2D shadowMaps[4] = { shadowMap2048x2048,
  shadowMap1024x1024, shadowMap512x512, shadowMap256x256 };
```

Such a numbering is similar to the numbering of mipmaps of color textures. Suppose we want to approximate an $m \times m$ PCF with an MRF based on nine

samples of a 3×3 PCF. The index of the shadow map, which can be used to get an adequate approximation, is computed as

```
float shadowMapIndex = log2( max(1.0, m/3.0) );
```

The value `shadowMapIndex` is a real number (e.g., the value `shadowMapIndex =2.415` corresponds to a 16×16 kernel), so we have to either round it toward the closest integer, or perform the following interpolation, which is similar to the one used in standard trilinear filtering:

```
float shadowIntensity = lerp(
   PCF(shadowMaps[ floor(shadowMapIndex) ], uv, d, 9),
   PCF(shadowMaps[ ceil(shadowMapIndex) ], uv, d, 9),
   frac(shadowMapIndex) );
```

Pros and cons. MRF enables creating large penumbrae with only a few depth texture samples. Since it is based on a small number of samples, it allows the computiation of shadow intensities using relatively complex filter kernels, which can produce continuous values without the noise inherent to plain PCF kernels with a low number of samples. As a result, MRF does not require any type of postfiltering. Compared to prefiltering methods, (e.g. [Donnelly and Lauritzen 06], [Annen et al. 07], [Annen et al. 08]), MRF does not introduce approximation-specific artifacts, such as light leaking, ringing, or precision deficiencies. Moreover, since MRF is based on a regular PCF, it is possible to utilize existing hardware features, for example hardware bilinear PCF or double-speed depth-only rendering.

2.2 Application to Adaptive Shadow Maps

Adaptive shadow maps (ASM) is a method that addresses perspective and projection aliasing that occurs in standard shadow mapping [Fernando et al. 01]. Another notable advantage of ASM is the ability to exploit frame-to-frame coherency, which was the main reason for implementing the method in Funcom's *Age of Conan*. The algorithm is shown schematically in Figure 2.3. The major steps of the algorithm are explained below.

Creating tiles hierarchy. We start by projecting the view frustum onto the near plane of the light's frustum. The projected frustum is clipped against a grid defined on this plane. This grid is view independent and static (does not change from frame to frame). Each grid cell intersecting with the frustum is called a *tile*. If a tile is closer to the projected frustum's top than a certain threshold distance, we subdivide it into four equal cells (in a quadtree-like fashion). Each resulting cell is tested against the frustum, and the cells intersecting with it are called *child tiles*. We continue the subdivision process to obtain a hierarchy of tiles as shown in Figure 2.3. Note that unlike for example, the cascaded shadow

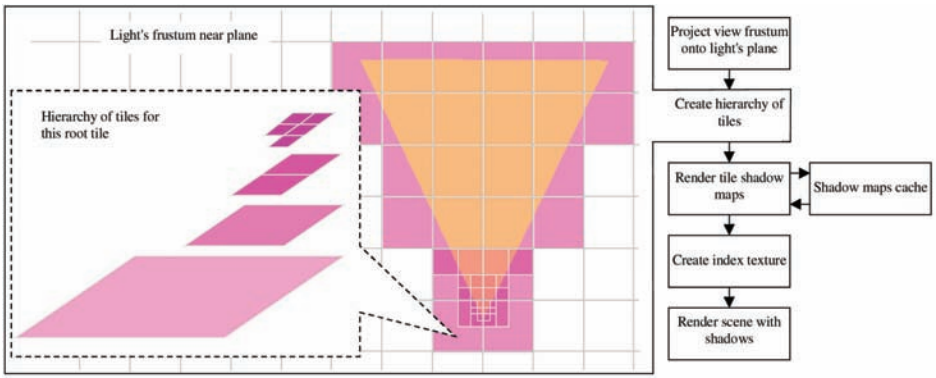


Figure 2.3. Adaptive Shadow Maps.

maps algorithm, here we do *not* subdivide the frustum itself; instead we use the frustum to determine which grid cells it intersects.

Shadow maps rendering. We render a fixed-resolution shadow map (e.g., 256×256) for every tile. Shadow maps are allocated from the single atlas (we use 4096×2048 depth texture in *Age of Conan*). Normally, there could be many tiles (about 100 in *Age of Conan*), but due to frame-to-frame coherency there is a high probability that a given shadow map was already rendered during previous frames. This is controlled through a shadow maps cache with LRU caching policy, (i.e., if the shadow map for a new tile is not present in the cache, then the shadow map which has not been in use for the largest number of frames will be reassigned to this tile).

Scene rendering. Since all shadow maps are equal in size, the shadow map sampling function requires knowledge only of the shadow map's offset within the atlas. These offsets are stored in a small dynamic index texture (we use a 128×128 texture). One can think of index texture as a standard shadow map that covers the entire view range, but instead of depth values, each texel contains the offsets, which are used to locate the actual tile shadow map in the atlas. Shadow map sampling code is given in Listing 2.1. As one can see, the only difference is index-texture indirection.

Pros and cons. ASM enables rendering of highly detailed shadows. Similar shadow mapping quality can be achieved only with standard shadow mapping when using a shadow map of very high resolution (for example, the resolution of an equivalent shadow map in *Age of Conan* is 16384×16384). Unlike projection-modifying approaches, such as [Stamminger and Drettakis 02] or [Martin and Tan 04], ASM does not suffer from temporal aliasing (since the tile's shadow map projection

```
float standardShadowMapSampling(float4 samplePosition)
{
  float4 shadowMapCoords =
    mul(samplePosition, shadowProjectionMatrix);
  return PCF(shadowMapTexture, shadowMapCoords);
}

float shadowMapSamplingASM(float4 samplePosition)
{
  float4 indexCoords =
    mul(samplePosition, shadowProjectionMatrix);
  float3 offset = tex2D(indexTexture, indexCoords.xy);
  float2 C = float2(tileShadowMapSize/atlasSize, 1);
  float3 shadowMapCoords = indexCoords*C.xxy + offset;
  return PCF(atlasTexture, shadowMapCoords);
}
```

Listing 2.1. Standard shadow map sampling vs ASM sampling.

matrices are view independent) and offers an intuitive control over the shadow map's texel distribution.

However, ASM imposes certain restrictions on a scene's granularity. Even though such situations do not occur frequently, in some cases we might need to render a number of shadow maps (we note that in *Age of Conan* we typically render one tile per several frames). As an extreme example, consider a scene that consists of just one huge object; the cost for rendering N shadow maps will be N times the cost of rendering such a scene. On the other hand, imagine a scene that consists of objects so small that they always overlap with just one tile; in this case the cost for rendering N tile shadow maps will be less or equal to the cost of whole scene. Therefore, a preferred scene should consist of a large number of lightweight, spatially compact objects rather than a few big and expensive-to-render geometry chunks.

Provided that granularity of the scene is reasonably fine and there is a certain frame-to-frame coherency, ASM significantly reduces shadow map rendering costs compared to standard shadow mapping. In this regard, one can view ASM as a method for distributing the cost of rendering a single shadow map over multiple frames.

A fundamental shortcoming of ASM is its inability to handle animated objects because such objects require updating the shadow map with every frame, while ASM relies on a shadow maps cache that holds data created over many frames. Similarly, light's position or direction cannot be changed on a per-frame basis because such a change invalidates the cache. In *Age of Conan* we use a separate

1024×1024 shadow map for rendering dynamic objects. Such a shadow map normally contains only a few objects, so it is inexpensive to render. Moreover, one can apply prefiltering techniques, (e.g., variance shadow maps), which may otherwise be problematic from the viewpoint of performance or quality. MRF naturally applies to the hierarchy of shadow maps produced with ASM.

2.3 Soft Shadows with Variable Penumbra Size

PCF with a fixed kernel size does not yield realistic shadows because it produces shadows with fixed penumbrae size, while penumbra in physically correct shadows usually vary greatly across the scene. The larger the light's area is, the bigger the difference should be between shadows from small objects and big objects. Thus, the kernel sizes in PCF should also vary depending on the estimated sizes of penumbrae.

One method for estimating penumbra size is proposed in the *percentage-closer soft shadows* (PCSS) scheme [Fernando 05]. In this method the size of a PCF kernel is a linear function of distance between a shaded fragment and its nearest occluder. The process of estimating the distance, which is called *blocker search*, largely resembles PCF; the difference is that instead of averaging results of depth tests, during blocker search one averages weighted depth values. When a light's area is large, the blocker search has to be performed on a large number of shadow map texels and thus it becomes as expensive as PCF. Attempting to reduce the number of samples may cause visible discontinuities and noise in penumbrae.

We use a faster method to estimate the distance to the occluder. A *depth extent map* (DEM) is a texture that contains minimum and maximum depth values computed over a certain region of the shadow map, which is used for detection of penumbrae regions [Isidoro and Sander 06]. We take the DEM's minimum depth value for the depth of the occluder, thus avoiding expensive blocker search. Since minimum depth value is a noncontinuous function, its use leads to discontinuities in penumbrae (such as clearly visible curves, along which shadows are vastly different). We compute the DEM for a low-resolution shadow map, and then use a single bilinear texture fetch to obtain piecewise-linear depth. The drawback of such an approach is the additional space required for storing the DEM. However, the DEM can be stored at a lower resolution than the resolution of the shadow map.

Occluders fusion. The most outstanding defect of PCF-based soft shadows is incorrect occluder fusion. The larger the penumbra size is, the more the artifacts stand out (see e.g., Figure 2.5(a)). The main source of the problem, illustrated in Figure 2.4, is the inability of a single shadow map to capture information needed

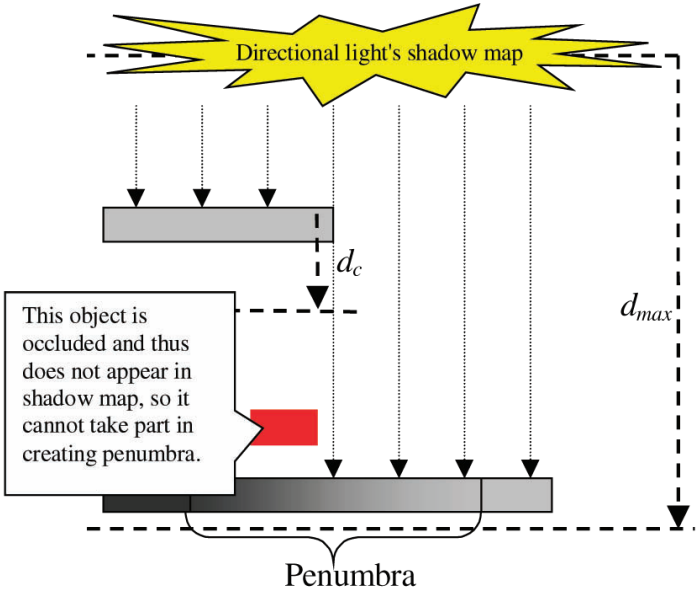


Figure 2.4. Shadow map layering.

to create shadows from area light. Each texel of a shadow map contains visibility information for a single light direction only, though light propagates along a range of directions.

This problem can be reduced relatively easily within the ASM framework. For a small set of tiles, which are closer to the viewer than a certain threshold distance, we render two shadow maps instead of one. First we create a regular shadow map and its corresponding DEM. Let $d_{\max}$ be the shadow map's range. Then, we create a *layer shadow map*, which is identical to the regular one, except that it is constructed using the fragments with depths within the range $[d + d_c, d_{\max}]$ (fragments with depth outside this range are discarded), where d is the corresponding minimum depth value from the DEM constructed for the regular shadow map, and d_c is a constant chosen by user. The penumbra over the scene's objects located within the range $[0; d + d_c]$ will be created using a regular shadow map only, thus occluder fusion will not be correct. However, one can use a layer shadow map to correct the penumbra on the objects located beyond $d + d_c$, as shown in Figure 2.5(b).

Shadow map layering significantly improved image quality in *Age of Conan*, removing a vast majority of occluder fusion artifacts. While theoretically one may utilize more than one layer, in *Age of Conan* one layer appeared to be sufficient. Adding more layers did not lead to any noticeable improvements.

(a) PCSS (from NVIDIA Direct3D SDK 10 Code Samples.)

(b) ASM + MRF + layering.

Figure 2.5. Occluders fusion: PCSS filters out penumbra details, ASM allows keeping them.

2.4 Results

We implemented our soft shadows algorithm in Funcom's MMORPG *Age of Conan*. Figure 2.6 shows in-game benchmark of two shadow mapping methods tested on Intel Core i7 2.66 MHz and AMD Radeon 4850.

Originally, the shadowing method in the released version of *Age of Conan* was standard shadow mapping, and cascaded shadow maps were added a year later with a patch. As shown in Figure 2.6, standard shadow mapping resulted in approximately 30% frame rate drop. The cascaded shadow map performance

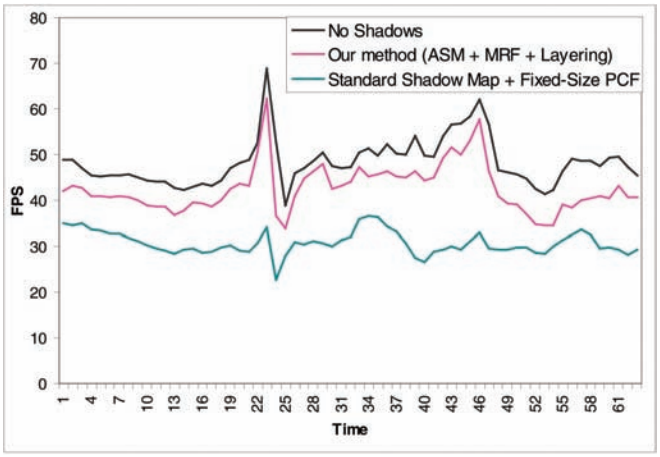


Figure 2.6. Benchmark: fly-by over a level from Age of Conan.

(not shown here) was worse. Implementing ASM-based soft shadows provided not only a substantial increase in image quality, but also a significant performance boost. We use ASM (with MRF and layering) to create shadows from static objects, and a separate 1024×1024 shadow map for dynamic objects, which is filtered with a fixed-size 3×3 PCF kernel.

Bibliography

[Annen et al. 07] Thomas Annen, Tom Mertens, P. Bekaert, Hans-Peter Seidel, and Jan Kautz. "Convolution Shadow Maps." In *European Symposium on Rendering*, pp. 51–60. Aire-la-Ville, Switzerland: Eurographics Association, 2007.

[Annen et al. 08] Thomas Annen, Tom Mertens, Hans-Peter Seidel, Eddy Flerackers, and Jan Kautz. "Exponential Shadow Maps." In *GI '08: Proceedings of Graphics Interface 2008*, pp. 155–161. Toronto, Canada: Canadian Information Processing Society, 2008.

[Donnelly and Lauritzen 06] William Donnelly and Andrew Lauritzen. "Variance Shadow Maps." In *I3D '06: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pp. 161–165. New York: ACM, 2006.

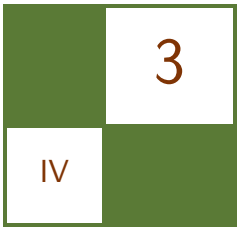
[Fernando et al. 01] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. "Adaptive Shadow Maps." In *SIGGRAPH '01: Proceedings of the 28th Annual Cconference on Computer Graphics and Interactive Techniques*, pp. 387–390. New York: ACM, 2001.

[Fernando 05] Randima Fernando. "Percentage-Closer Soft Shadows." In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, p. 35. New York: ACM, 2005.

[Isidoro and Sander 06] John R. Isidoro and Pedro V. Sander. "Edge Masking and Per-Texel Depth Extent Propagation For Computation Culling During Shadow Mapping." In *ShaderX^5: Advanced Rendering Techniques*. Hingham:MA: Charles River Media, 2006.

[Martin and Tan 04] Tobias Martin and Tiow-Seng Tan. "Anti-Aliasing and Continuity with Trapezoidal Shadow Maps." In *Proceedings of Eurographics Symposium on Rendering*, pp. 153–160. Aire-la-Ville, Switzerland: Eurographics Association, 2004.

[Stamminger and Drettakis 02] Marc Stamminger and George Drettakis. "Perspective Shadow Maps." In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 557–562. New York: ACM, 2002.



Adaptive Volumetric Shadow Maps

Marco Salvi, Kiril Vidimče, Andrew Lauritzen, Aaron Lefohn, and Matt Pharr

This chapter describes *adaptive volumetric shadow maps* (AVSM), a new approach for real-time shadows that supports high-quality shadowing from dynamic volumetric media such as hair and smoke. AVSMs compute approximate volumetric shadows for real-time applications such as games, for which predictable performance and a fixed, small memory footprint are required (and for which approximate solutions are acceptable).

We first introduced AVSM in a paper at the 2010 Eurographics Symposium on Rendering [Salvi et al. 10]; this chapter reviews the main ideas in the paper and details how to efficiently implement AVSMs on DX11-class graphics hardware. AVSMs are practical on today's high-end GPUs; for example, rendering Figure 3.4 requires 8.6 ms with opacity shadow maps (OSMs) and 12.1 ms with AVSMs—an incremental cost of 3.5 ms to both build the AVSM data structure and to use it for final rendering.

3.1 Introduction and Previous Approaches

Realistic lighting of volumetric and participating media such as smoke, fog, or hair adds significant richness and realism to rendered scenes. Self-shadowing provides important visual cues that define the shape and structure of such media. However, in order to compute self-shadowing in volumetric media, it is necessary to accumulate partial occlusion between visible points and light sources in the scene; doing so requires capturing the effect of all of the volumetric objects between two points and is generally much more expensive than computing shadows from opaque surfaces. As such, while it is common for offline renderers (e.g., those used in film rendering) to compute volumetric shadows, the computation and memory costs required to simulate light transport through participating

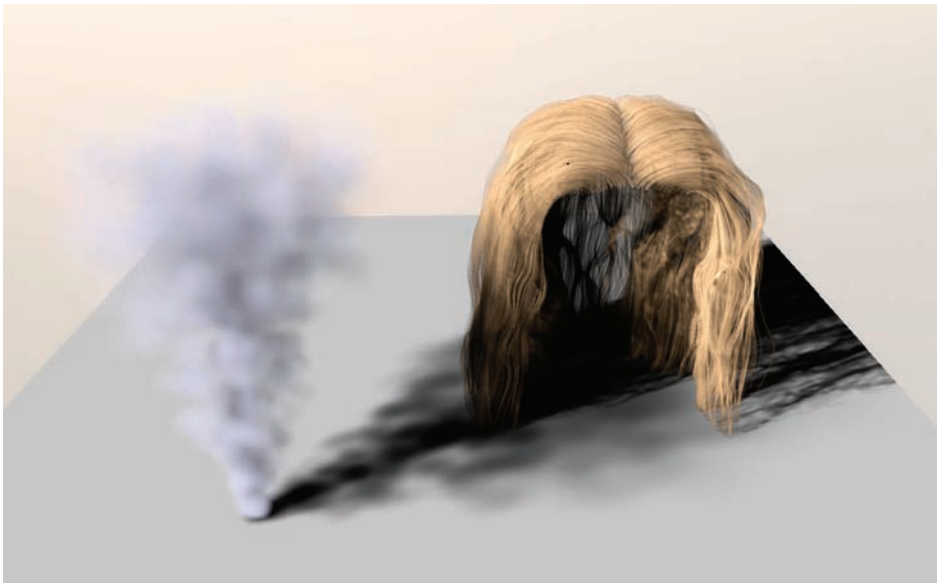


Figure 3.1. This image shows self-shadowing smoke and hair, both seamlessly rendered into the same adaptive volumetric shadow map. (Hair model courtesy of Cem Yuksel).

media have limited their use in real-time applications. Existing solutions for real-time volumetric shadowing exhibit slicing artifacts due to nonadaptive sampling, cover only a limited depth range, or are limited to one type of media (e.g., only hair, only smoke, etc.).

Adaptive shadow representations such as deep shadow maps have been used widely in offline rendering [Lokovic and Veach 00, Xie et al. 07]. Deep shadow maps store an adaptive, lossy-compressed representation of the visibility function for each light-space texel, though it is not clear how they can be implemented efficiently enough for real-time performance on today's graphics hardware, due to their high costs in term of storage and memory bandwidth.

Many volumetric shadowing techniques have been developed for interactive rendering. See our paper [Salvi et al. 10] for a detailed discussion of previous approaches; here,in this chapter, we will highlight the most widely known alternatives. A number of approaches discretize space into regularly spaced slices, for example opacity shadow maps [Kim and Neumann 01]. These methods typically suffer from aliasing, with variations specialized to handle small particles that can display view-dependent shadow popping artifacts even with static volumes [Green 08]. Deep opacity maps improve upon opacity shadow maps specifically for hair rendering by warping the sampling positions in the first depth

layer [Yuksel and Keyser 08]. Occupancy maps also target hair rendering and use regular sampling, but capture many more depth layers than opacity- or deep-opacity- shadow maps by using only one bit per layer. However, they are limited to volumes composed of occluders with identical opacity [Sintorn and Assarson 09]. Mertens et al. describe a fixed-memory shadow algorithm for hair that adaptively places samples based on a k-means clustering estimate of the transmittance function, assuming density is uniformly distributed within a small number of clusters [Mertens et al. 04]. Recently, Jansen and Bavoil introduced Fourier opacity mapping, which addresses the problem of banding artifacts, but where the detail in shadows is limited by the depth range of volume samples along a ray and may exhibit ringing artifacts [Jansen and Bavoil 10]. Finally, Enderton et al. [Enderton et al. 10] have introduced a technique for handling all types of transparent occluders in a fixed amount of storage for both shadow and primary visibility, generating a stochastically sampled visibility function, though their approach requires a large number of samples for good results.

AVSM generates an adaptively sampled representation of the volumetric transmittance in a shadow-map-like data structure, where each texel stores a compact approximation of the transmittance curve along the corresponding light ray. AVSM can capture and combine transmittance data from arbitrary dynamic occluders, including combining soft media like smoke and well-localized denser media such as hair. It is thus both a versatile and a robust approach, suitable for handling volumetric shadows in a variety of situations in practice. The main innovation introduced by AVSM is a new, streaming lossy compression algorithm that is capable of building a constant-storage, variable-error representation of visibility for later use in shadow lookups.

3.2 Algorithm and Implementation

Adaptive volumetric shadow maps encode the fraction of visible light from the light source over the interval $[0, 1]$ as a function of depth at each texel. This quantity, the *transmittance*, is the quantity needed for rendering volumetric shadows. It is defined as

$$t(z) = \mathrm{e}^{-\int_0^z f(x)\,\mathrm{d}x}, \tag{3.1}$$

where $f(x)$ is an attenuation function that represents the amount of light absorbed or scattered along a light ray (see Figure 3.2).

The AVSM representation stores a fixed-size array of irregularly placed samples of the transmittance function. Array elements, the *nodes* of the approximation, are sorted front-to-back, with each node storing a pair of depth and transmittance values (d_i, t_i). Because we adaptively place the nodes in depth, we can represent a rich variety of shadow blockers, from soft and transmissive particles to sharp and opaque occluders. The number of nodes stored per texel is a user-defined quantity, the only requirement being that we store two or more

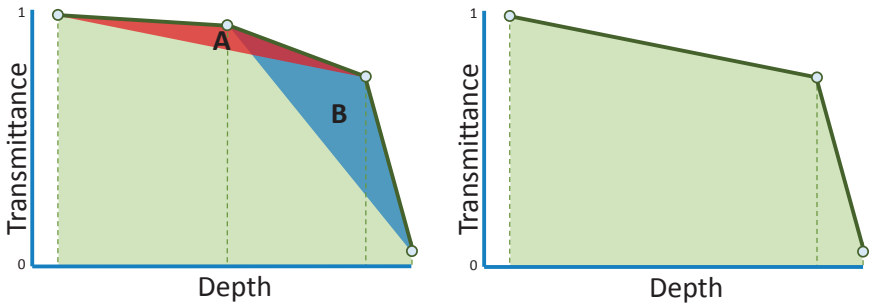


Figure 3.2. AVSM computes a compressed representation of transmittance along a light ray using an area-based curve simplification scheme. This figure depicts compressing a 4-node curve to three nodes. The algorithm removes the node that results in the smallest change in area under the curve, determined by computing the area of the triangle created between the candidate node and its adjacent neighbors (triangles A and B). The right figure shows that we remove the second node from the left because triangle A is smaller than triangle B.

nodes per texel. More nodes allow for a better approximation of transmittance and higher quality shadows, but at the expense of increased storage and computational costs. We have found that, in practice, 8–12 nodes (a cost of 64–96 bytes per texel in the AVSM when full precision is used) give excellent results.

In HLSL code, our AVSM nodes are implemented with a simple structure:

```
#define AVSM_NODE_COUNT 8
#define AVSM_RT_COUNT (AVSM_NODE_COUNT / 4)

struct AVSMData
{
    float4 depth[AVSM_RT_COUNT];
    float4 trans[AVSM_RT_COUNT];
};
```

To simplify our implementation we always store a multiple of four nodes in the AVSM. A group of four nodes fills two render targets (one for depth and one for transmittance). This requirement sets a limit of sixteen nodes per texel since current graphics APIs don't allow binding of more than eight render targets at once.

The following sections will describe first how we generate samples of the full transmittance function, then how these samples are compressed into the AVSM representation, and finally how they are interpolated at rendering time in shading calculations.

3.2.1 Capturing Fragments of Transparent Objects

The first step in the implementation is to render from the light source and capture a per-pixel linked list of all of the transparent fragments visible from the camera at each pixel[1]. A subsequent pass, described in the Section 3.2.3, compresses these linked lists to the compact AVSM representation.

In a manner similar to that of standard shadow maps, AVSMs are created by rendering the scene from the light's viewpoint. AVSM can handle both opaque objects and objects of varying thickness and density. We need to store all of the transparent fragments up to the first opaque fragment, as well as the first opaque fragment itself (if there is one). This information is enough to exactly specify the transmittance curve as seen from the light source. (Transparent fragments behind the first opaque one, as well as any subsequent opaque fragments are irrelevant, since the first opaque fragment immediately drives the transmittance to zero for all depths behind it.)

While it is not strictly necessary, in order to speed up the creation of a volumetric shadow map, we may begin by rendering the opaque objects into a depth buffer to establish the depth beyond which we don't need to capture transparent fragments. We then render a pass that includes the transparent objects, using the already-computed depth buffer for z-tests (with a less-than-or-equal-to test mode), but with z-writes disabled. Each fragment that passes this test is added to a per-pixel linked list of fragments.

The per-pixel linked lists of light-attenuating segments are captured using DX11's support for atomic gather/scatter memory operations in pixel shaders [Yang et al. 10]. The basic idea is that a first r/w buffer is allocated with integer pixel values, each pixel representing a pointer to the head of a list and initially storing a "nil" value (e.g., -1.) These pointers encode offsets into a second larger r/w buffer which stores all lists nodes. A pixel shader then allocates a node from the second buffer by atomically incrementing a globally shared counter (see DirectX11 UAV counters) whose value represents a pointer to a new node. It then inserts at the head of the list the newly allocated node by atomically swapping the previous head pointer with the new one. The content of the new node is updated and its next pointer is set to the previous head of the list.

For our test scenes no more than 20 MB of this buffer is typically needed. Listing 3.1 shows HLSL code for creating these per-pixel linked lists.

[1]While AVSM is designed to be a streaming compression algorithm, such an implementation requires support for read-modify-write framebuffer operations in the pixel shader. DirectX11 adds the ability to perform unordered read-modify-write operations on certain buffer types in the pixel shader; however, for AVSM's transmittance-curve-simplification algorithm we need to ensure that each pixel's framebuffer memory is modified by only one fragment at a time (per-pixel lock). Because current DX11 HLSL compilers forbid per-pixel locks, we implement AVSM with variable-memory version that uses the current DX11 rendering pipeline to first capture all transparent fragments seen from the light and then compressing them into the AVSM representation.

```
#define MAX_BUFFER_NODES (1<<24)

RWStructuredBuffer<ListTexNode> gListTexSegmentNodesUAV;
RWTexture2D<uint>               gListTexFirstNodeAddrUAV;

// Allocate a generic node
bool AllocNode(out uint newNodeAddress)
{
    // alloc a new node
    newNodeAddress = gListTexNodesUAV.IncrementCounter();

    // running out of memory?
    return newNodeAddress < MAX_BUFFER_NODES;
}

// Insert a new node at the head of the list
void InsertNode(in int2 screenAddress,
                in uint newNodeAddress,
                in ListTexNode newNode)
{
    uint oldNodeAddress;
    InterlockedExchange(gListTexFirstNodeAddrUAV[screenAddress],
                        newNodeAddress,
                        oldNodeAddress);

    newNode.next = oldNodeAddress;
    gListTexNodesUAV[newNodeAddress] =  newNode;
}
```

Listing 3.1. Node allocation and insertion code of a generic list (DirectX11/Shader Model 5).

This process can also be used with transparent objects with finite extent and uniform density in depth—not just billboards. Each object's fragment can store in the list start and end points along the corresponding ray from the light source to define a segment, along with exit transmittance (entry transmittance is implicitly assumed to be set to 1). For example, given billboards representing spherical particles, we insert a segment representing the ray's traversal through the particle; for hair we insert a short segment where the light enters and exits the hair strand; for opaque blockers, we insert a short, dense segment that takes the transmittance to zero at the exit point.

3.2.2 AVSM Compression

After the relevant fragments have been captured in a list at each pixel, we need to compute a compressed transmittance curve in a fixed amount of memory. Doing

so not only saves a considerable amount of storage and bandwidth, but makes lookups very efficient. In general, the number of transparent fragments at a pixel will be much larger than the number of AVSM nodes (e.g., Figure 3.6 shows a transmittance curve with 238 nodes and its 12 nodes counterpart compressed with AVSM), therefore, we use a streaming compression algorithm that in a single (post-list-creation) rendering pass approximates the original curve.

Each node of our piecewise transmittance curve maps to an ordered sequence of pairs (d_i, t_i) that encode node position (depth) along a light ray and its associated transmittance. AVSMs store the transmittance curve as an array of depth-transmittance pairs (d_i, t_i) using two single-precision floating-point values[2]. An important ramification of our decision to use a fixed, small number of nodes is that the entire compressed transmittance curve can fit in on-chip memory during compression. As with classic shadow maps we clear depth to the far plane value, while transmittance is set to 1 in order to represent empty space.

We insert each new occluding segment by viewing it as a compositing operation between two transmittance curves, respectively representing the incoming blocker and the current transmittance curve. Given two light blockers, A and B, located along the same light ray, we write the density function $f_{AB}(x)$ as a sum of their density functions $f_A(x)$ and $f_B(x)$. By simply applying Equation eqrefeq:transmittance we can compute their total transmittance:

$$\begin{aligned} t_{\text{tot}}(z) &= \mathrm{e}^{-\int_0^z f_{AB}(x)\,\mathrm{d}x} \\ &= \mathrm{e}^{-\int_0^z f_A(x)\,\mathrm{d}x}\mathrm{e}^{-\int_0^z f_B(x)\,\mathrm{d}x} = t_A(z)t_B(z). \end{aligned} \tag{3.2}$$

In the absence of lossy compression, the order of composition is not important. More relevantly, this equation shows that the resulting total transmittance is given by the product of the two transmittance functions respectively associated to each light blocker.

Conceptually, our task is to compute a set of nodes that accurately represents the full transmittance curve.[3] For the first few segments in the linked list, up to the fixed number of AVSM nodes that we are storing, our task is straightforward: we insert the segment into the AVSM array. We move the nodes with depths greater than the new ones (a segment maps to two nodes) one element forward, making an opening at the offset into the array, at which the new nodes should be added, and creating new transmittance values for each opening by linearly interpolating neighboring nodes. We then composite this curve with the curve represented by the incoming segment (see Equation (3.2)). Once the (on-chip) array of nodes contains more nodes than can be stored in the AVSM texture,

[2]It's also possible to use half-precision values for relatively simple scenes without incurring noticeable image artifacts.

[3]The problem of approximating a polygonal curve P by a simpler polygonal curve Q is of interest in many fields and has been studied in cartography, computer graphics, and elsewhere; see our EGSR paper for more references on this topic.

we apply a curve simplification algorithm; our method compresses transmittance data simply by removing the node that contributes the least to the overall transmittance curve shape. In other words, we remove the node, that once removed, generates the smallest variation to curve integral (see Figure 3.2). Compression proceeds by removing one node at a time until the maximum node count is reached. We apply compression only to internal nodes. In practice, this is a benefit because these uncompressed nodes provide important visual cues such as transition into a volume or the shadows cast from a volume onto opaque surfaces.

Although transmittance varies exponentially between nodes, like deep shadow maps, we assume linear variation to simplify area computations. This allows us to write the transmittance integral I_t for an N node curve as the sum of $N-1$ trapezoidal areas:

$$I_t = \sum_{i=0}^{N-1} \frac{(d_{i+1} - d_i)(t_i + t_{i+1})}{2}.$$

The removal of an internal ith node affects only the area of the two trapezoids that share it. Since the rest of the curve is unaffected we compute the variation of its integral $\triangle_{t_i}$ with a simple, geometrically derived formula:

$$\triangle_{t_i} = |(d_{i+1} - d_{i-1})(t_{i+1} - t_i) - (d_{i+1} - d_i)(t_{i+1} - t_{i-1})|.$$

In practice, due to the lossy compression, the order in which segments are inserted can affect the results. In particular, when generating the per-pixel linked lists in the previous pass, the parallel execution of pixel shaders inserts segments into the linked lists in an order that may vary per-frame even if the scene and view are static. Inconsistent ordering can result in visible temporal artifacts, although they are mostly imperceptible in practice when using eight or more AVSM nodes or when the volumetric media is moving quickly (e.g., billowing smoke). In those rare cases when a consistent ordering cannot be preserved and the number of nodes is not sufficient to hide these artifacts, it is also possible to sort the captured segments by depth via an insertion sort before inserting them. We discuss the cost of this sort in Section 3.3.3.

3.2.3 AVSM Sampling

Sampling AVSMs can be seen as a generalization of a standard shadow-map depth test [Williams 78] of translucent occluders. Instead of a binary depth test, we evaluate the transmittance function at the receiver depth[4].

Due to the irregular and nonlinear nature of the AVSM data, we cannot rely on texture-filtering hardware, and we implement filtering in programmable shaders. For a given texel, we perform a search over the entire domain of the curve to find

[4]It's also possible to render opaque blockers into an AVSM. In this case AVSM sampling will behave exactly as the method introduced by [Williams 78].

the two nodes that bound the shadow receiver of depth d, we then interpolate the bounding nodes' transmittance (t_l, t_r) to intercept the shadow receiver.

In order to locate the two nodes that bound the receiver depth (i.e., a segment), we use a fast two-level search; since our representation stores a fixed number of nodes, memory accesses tend to be coherent and local, unlike with variable-length linked-list traversals necessary with techniques like deep shadow maps [Lokovic and Veach 00]. In fact, the lookups can be implemented entirely with compile-time (static) branching and array indexing, allowing the compiler to keep the entire transmittance curve in registers. Listing 3.3 shows an implementation of our AVSM segment-finding algorithm specialized for an eight node visibility curve, which is also used for both segment insertion and sampling/filtering[5].

As we do at segment-insertion time, we again assume space between two nodes to exhibit uniform density, which implies that transmittance varies exponentially between each depth interval (see Equation (3.1)), although we have found linear interpolation to be a faster and visually acceptable alternative:

$$T(d) = t_l + (d - d_l) \cdot \frac{t_r - t_l}{d_r - d_l}$$

This simple procedure is the basis for point filtering. Bilinear filtering is straightforward; the transmittance $T(d)$ is evaluated over four neighboring texels and linearly weighted.

```
struct AVSMSegment
{
    int   index;
    float depthA;
    float depthB;
    float transA;
    float transB;
};

AVSMSegment FindSegmentAVSM8(in AVSMData data,
                             in float receiverDepth)
{
    AVSMSegment Output;
    int         index;
    float4      depth, trans;
    float       leftDepth, rightDepth, leftTrans, rightTrans;
```

[5]Please see the accompanying demo source code for a generalized implementation that supports 4-, 8-, 12- and 16-node AVSM textures.

```
    // We start by identifying the render target that..
    // ..contains the nodes we are looking for..
    if (receiverDepth > data.depth[0][3]) {
        depth       = data.depth[1];
        trans       = data.trans[1];
        leftDepth   = data.depth[0][3];
        leftTrans   = data.trans[0][3];
        rightDepth  = data.depth[1][3];
        rightTrans  = data.trans[1][3];
        Output.index = 4;
    } else {
        depth       = data.depth[0];
        trans       = data.trans[0];
        leftDepth   = data.depth[0][0];
        leftTrans   = data.trans[0][0];
        rightDepth  = data.depth[1][0];
        rightTrans  = data.trans[1][0];
        Output.index = 0;
    }
    // ..we then look for the exact nodes that wrap..
    // ..around the shadow receiver.
    if (receiverDepth <= depth[0]) {
        Output.depthA = leftDepth;
        Output.depthB = depth[0];
        Output.transA = leftTrans;
        Output.transB = trans[0];
    } else if (receiverDepth <= depth[1]) {
        ....
        ....
    } else {
        Output.index += 4;
        Output.depthA = depth[3];
        Output.depthB = rightDepth;
        Output.transA = trans[3];
        Output.transB = rightTrans;
    }
    return Output;
}
```

Listing 3.3. Segment finding code for 8-node AVSM data.

3.3 Comparisons

We have compared AVSM to a ground-truth result, deep shadow maps (DSM), Fourier opacity maps (FOM), and opacity shadow maps (OSM). All techniques were implemented using the DirectX11 rendering and compute APIs.

All results are gathered on an Intel Core i7 quad-core CPU running at 3.33 GHz running Windows 7 (64-bit) and an ATI Radeon 5870 GPU.

```
[unroll] for (i = 0; i < AVSM_NODE_COUNT + 2; ++i) {
    // Compute render target and vector element indices
    const int rtIdx = i >> 2;
    const int elemIdx = i & 0x3;

    float tempDepth, tempTrans;
    // Insert last segment node
    [flatten]if (i == postMoveSegmentEndIdx) {
        tempDepth = segmentDepth[1];
        tempTrans = newNodesTransOffset[1];
    // Insert first segment node
    } else if (i == postMoveSegmentStartIdx) {
        tempDepth = segmentDepth[0];
        tempTrans = newNodesTransOffset[0];
    // Update all nodes in between the new two nodes
    } else if ((i > postMoveSegmentStartIdx) &&
               (i < postMoveSegmentEndIdx)) {
        tempDepth = depth[i-1];
        tempTrans = trans[i-1];
    // Update all nodes located behind the new two nodes
    } else if ((i > 1) && (i > postMoveSegmentEndIdx)) {
        tempDepth = depth[i-2];
        tempTrans = trans[i-2];
    // Update all nodes located in front the new two nodes
    } else {
        tempDepth = depth[i];
        tempTrans = trans[i];
    }

    // Linearly interpolates transmittance along the incoming..
    // ..segment and composite it with the current curve
    tempTrans *= Interp(segmentDepth[0], segmentDepth[1],
                        FIRST_NODE_TRANS_VALUE,
                        segmentTransmittance, tempDepth);

    // Generate new nodes
    newDepth[rtIdx][elemIdx] = d;
    newTrans[rtIdx][elemIdx] = t;
}
```

Listing 3.2. Segment insertion code for AVSMs. Note that there is no dynamic branching nor dynamic indexing in this implementation, which makes it possible for intermediate values to be stored in registers and for efficient GPU execution.

Figure 3.3. A comparison of smoke added to a scene from a recent game title with AVSM with 12 nodes (left) and deep shadow maps (right). Rendering the complete frame takes approximately 32 ms, with AVSM generation and lookups consuming approximately 11 ms of that time. AVSM is 1–2 orders of magnitude faster than a GPU implementation of deep shadow maps and the uncompressed algorithm, yet produce a nearly identical result. (Thanks to Valve Corporation for the game scene.)

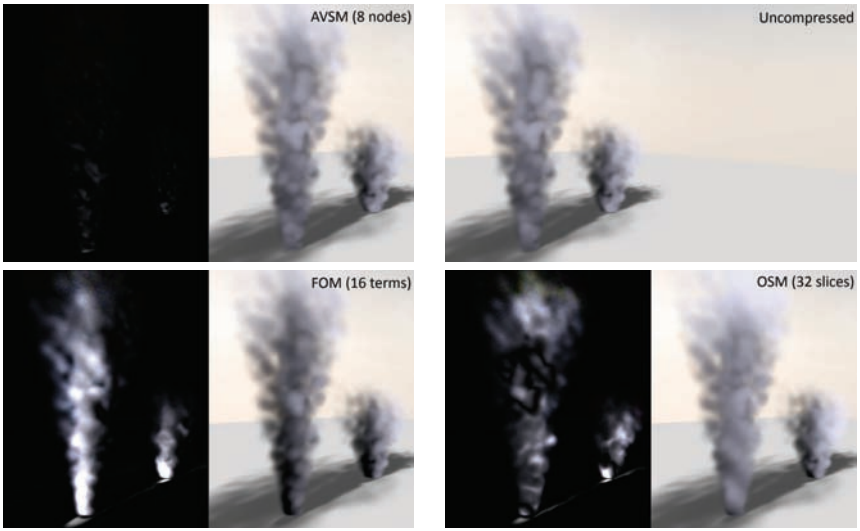


Figure 3.4. Comparison of AVSM, Fourier opacity maps, and opacity shadow maps to the ground-truth uncompressed result in a scene with three separate smoke columns casting shadows on each other: AVSM with eight nodes (top left), ground-truth uncompressed (top right), Fourier opacity maps with 16 expansion terms (bottom left), and opacity shadow maps with 32 slices (bottom right). Note how closely AVSM matches the ground-truth image. While the artifacts of the other methods do not appear problematic in these still images, the artifacts are more apparent when animated. Note that the differerent images have been enhanced by 4x to make the comparison more clear.

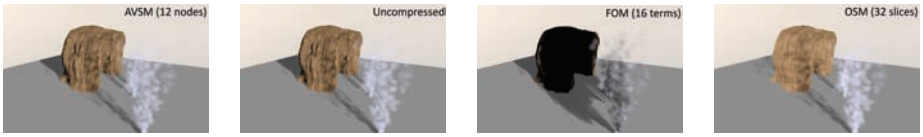


Figure 3.5. This scene compares (from left to right) AVSM (12 nodes), uncompressed, opacity shadow maps (32 slices), and Fourier opacity maps (16 expansion terms). Note that AVSM-12 and uncompressed are nearly identical and the other methods show substantial artifacts. In particular FOM suffers from severe over-darkening/ringing problems generated by high-frequency light blockers like hair and by less-than-optimal depth bounds. Also note that these images use only bilinear shadow filtering. Using a higher-quality filtering kernel substantially improves the shadow quality.

3.3.1 Qualitative Evaluation

Figure 3.3 shows AVSMs (12 nodes) compared to deep shadow maps (error threshold set to 0.002). There is little perceptible difference between the results, demonstrating that for this real-time scene, our decision to permit variable error per texel is not a problem. The accuracy of AVSM is further validated by inspecting the transmittance curves and seeing that even with eight nodes, AVSM very closely approximates the true transmittance curves. The results for sampling the uncompressed data also look identical. Our experience is that eight nodes results in acceptable visual quality for all views and configurations in this scene. All shadow map sizes in these images are 256^2.

Figure 3.4 shows a visual comparison among 8-node AVSM, 16-term Fourier opacity maps, and 32-slice opacity shadow maps against the ground-truth uncompressed result for a scene with three smoke columns casting shadows on each other. Note how much more closely the AVSM matches the ground-truth uncompressed result. The quality improvements are especially noticeable when animated. A key benefit of AVSM compared with these other real-time methods is that AVSM quality is much less affected by the depth range covered by the volumetric occluders.

3.3.2 Quantitative Evaluation

We validate the AVSM compression algorithm accuracy by inspecting a number of transmittance curves and comparing to the ground-truth uncompressed data as well as the deep shadow map compression technique. Overall, we see that the 4-node AVSM shows significant deviations from the correct result, 8-node AVSM matches closely with a few noticeable deviations, and 12-node AVSM often matches almost exactly.

Figure 3.6 shows a transmittance curve from a combination of smoke and hair (see image in Figure 3.5) with discrete steps for each blonde hair and smooth

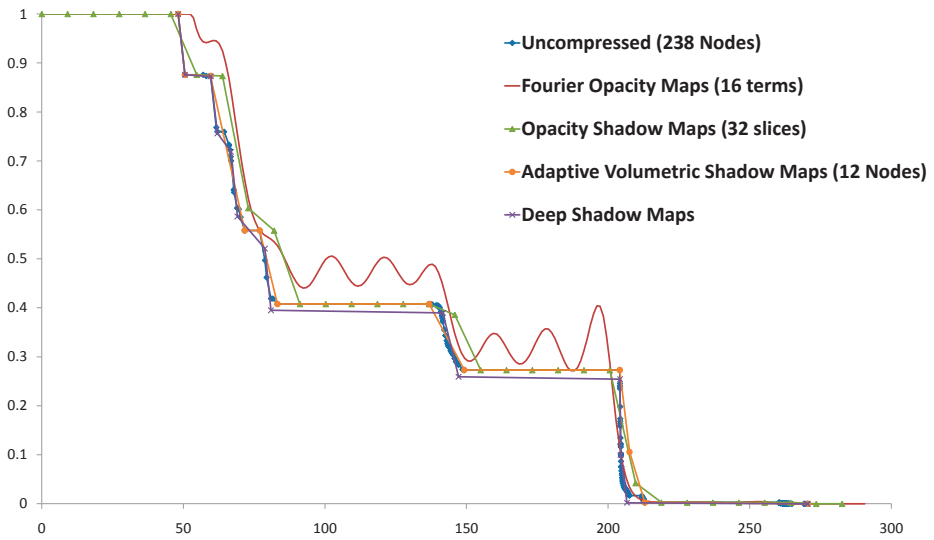


Figure 3.6. Transmittance curves computed for a scene with a mix of smoke and hair for AVSM (12 nodes) and the ground-truth uncompressed data (238 nodes). The hairs generate sharp reductions in transmittance, whereas the smoke generates gradually, decreasing transmittance. AVSM matches the ground-truth data much more closely than the other real-time methods.

transitions in the smokey regions. Note that the 12-node AVSM matches the ground-truth data much more closely than the opacity or Fourier shadow map (both of which use more memory than AVSM to represent shadow data) and is similar to the deep shadow map but uses less memory and is 1–2 orders of magnitude faster.

3.3.3 Performance and Memory

AVSM achieves its goal of adaptively sampling volumetric transmittance curves with performance high enough for real-time rendering throughout the Valve software scene (Figure 3.3). Table 3.1 shows the performance results for the view shown in Figure 3.4 for AVSM compared to opacity shadow maps, Fourier opacity maps, deep shadow maps, and the uncompressed approach. For this scene, AVSM compression takes only 0.5–1.5 ms, AVSM lookups take 3–10 ms depending on the number of AVSM nodes, capturing the segments takes 0.4 ms, and sorting

	AVSM4	AVSM8	AVSM16	OSM32	FOM16	DSM
Compress	0.5 ms	0.7 ms	1.6 ms	1 ms	1.1 ms	193 ms
Filtering	3 ms	5.4 ms	9.5 ms	1.4 ms	8.9 ms	52 ms
Total Time	9.7 ms	12.1 ms	17.43 ms	8.6 ms	15 ms	251 ms
Mem Usage	22(2)MB	24(4)MB	28(8)MB	8 MB	8 MB	40 MB

Table 3.1. Performance and memory results for 256^2 resolution, adaptive volumetric shadow maps (AVSM) with 4, 8 and 16 nodes, opacity shadow maps (OSM) with 32 slices, Fourier opacity maps (FOM) with 16 expansion terms, deep shadow maps (DSM), and the ground-truth uncompressed data for the scene shown in Figure 3.4. The AVSM compression algorithm takes 0.5–1.6 ms to build our representation of the transmittance curve even when there are hundreds of occluders per light ray. The total memory required for AVSM and DSM implementations on current graphics hardware is the size of the buffer used to capture the occluding segments plus the size of the compressed shadow map (shown in parentheses).

the segments (via insertion sort) before compression takes 3 ms[6]. As discussed earlier, the errors arising from not sorting are often imperceptible so sorting can usually be skipped—reducing the AVSM render-time to be nearly identical to that of opacity and Fourier opacity maps.

There are two key sources to AVSM performance. First is the use of a streaming compression algorithm that permits direct construction of a compressed transmittance representation without first building the full uncompressed transmittance curve. The second is the use of a fixed, small number of nodes such that the entire representation can fit into on-chip memory. While it may be possible to create a faster deep shadow map implementation than ours, sampling deep shadow maps' variable-length linked lists is costly on today's GPUs, and it may result in low SIMD efficiency. In addition, during deep shadow map compression, it is especially challenging to keep the working set entirely in on-chip memory.

Table 3.1 also shows the memory usage for AVSM, deep shadow maps, and the uncompressed approach for the smoke scene shown in Figure 3.4. Note that the memory usage for the variable-memory algorithms shows the amount of memory allocated, not the amount actually used per frame by the dynamically generated linked lists.

3.4 Conclusions and Future Work

Adaptive volumetric shadow maps (AVSM) provide an effective, flexible, and robust volumetric shadowing algorithm for real-time applications. AVSMs achieve a high level of performance using a curve-simplification compression algorithm

[6]While the AVSM blockers, insertion, and sampling code have received much attention, we don't currently have an optimized list sorting implementation but we expect it is possible to do significantly better than our current method.

that supports directly building the compressed transmittance function on-the-fly while rendering. In addition, AVSM constrains the compressed curves to use a fixed number of nodes, allowing the curves to stay in on-chip memory during compression. As the gap between memory bandwidth and compute capability continues to widen, this characteristic of the algorithm indicates that it is likely to scale well with future architectures.

One limitation of AVSM is the introduction of variable error per texel in exchange for the speed and storage benefits of fixed storage and fast compression. While we show in our test scenes and analysis that this is a valuable trade-off to make for real-time applications insofar as it affords high performance and rarely produces perceptible artifacts, offline rendering users that need absolute quality guarantees may want to continue to use a constant-error compression strategy such as deep shadow maps.

A second limitation is that implementations using current real-time graphics pipelines require a potentially unbounded amount of memory to first capture all occluding segments along all light rays. In addition, the unordered concurrency in pixel shaders means that when working with a low number of AVSM nodes per texel the segments may need to be re-sorted after capture to eliminate certain temporal artifacts. If future graphics pipelines support read-modify-write memory operations with a stable order, such as ordering by primitive ID, this limitation will go away.

Moreover, while our implementation requires DirectX11-compliant hardware, it is interesting to note that sampling and filtering an AVSM requires only DirectX9-compliant hardware. Moreover, since in many cases volumetric shadows exhibit very low spatial frequency and require low resolution shadow maps, it should be possible to implement AVSM on a current game console like Sony PS3® by writing a specialized SPU-based software rasterizer for particles/billboards that build AVSMs on-chip and in a single pass, avoiding locks and per-pixel lists, with AVSM sampling and filtering left to the GPU.

3.5 Acknowledgments

We thank Jason Mitchell and Wade Schin from Valve Software for the Left-for-Dead-2 scene and their valuable feedback; and Natasha Tatarchuk and Hao Chen from Bungie and Johan Andersson from DICE for feedback on early versions of the algorithm. Thanks to the the entire Advanced Rendering Technology team, Nico Galoppo and Doug McNabb at Intel for their contributions and support. We also thank others at Intel: Jeffery Williams and Artem Brizitsky for help with art assets; and Craig Kolb, Jay Connelly, Elliot Garbus, Pete Baker, and Mike Burrows for supporting the research.

Bibliography

[Enderton et al. 10] Eric Enderton, Erik Sintorn, Peter Shirley, and David Luebke. "Stochastic Transparency." In *I3D '10: Proceedings of the 2010 Symposium on Interactive 3D Graphics and Games*, pp. 157–164. New York: ACM, 2010.

[Green 08] Simon Green. "Volumetric Particle Shadows." http://developer.download.nvidia.com/compute/cuda/sdk/website/C/src/smokeParticles/doc/smokeParticles.pdf, 2008.

[Jansen and Bavoil 10] Jon Jansen and Louis Bavoil. "Fourier Opacity Mapping." In *I3D '10: Proceedings of the 2010 Symposium on Interactive 3D Graphics and Games*, pp. 165–172. New York: ACM, 2010.

[Kim and Neumann 01] Tae-Yong Kim and Ulrich Neumann. "Opacity Shadow Maps." In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pp. 177–182. Aire-la-Ville, Switzerland: Eurographics Assocaition, 2001.

[Lokovic and Veach 00] Tom Lokovic and Eric Veach. "Deep Shadow Maps." In *Proceedings of ACM SIGGRAPH 2000, Computer Graphics Proceedings, ACS*, pp. 385–392. New York: ACM, 2000.

[Mertens et al. 04] Tom Mertens, Jan Kautz, Philippe Bekaert, and F. van Reeth. "A Self-Shadowing Algorityhm for Dynamic Hair using Clustered Densities." In *Rendering Techniques 2004: Eurographics Symposium on Rendering*. Aire-la-Ville, Switzerland: Eurographics, 2004.

[Salvi et al. 10] Marco Salvi, Kiril Vidimče, Andrew Lauritzen, and Aaron Lefohn. "Adaptive Volumetric Shadow Maps." In *Eurographics Symposium on Rendering*, pp. 1289–1296. Aire-la-Ville, Switzerland: Eurographics Association, 2010.

[Sintorn and Assarson 09] Erik Sintorn and Ulf Assarson. "Hair Self Shadowing and Transparency Depth Ordering Using Occupancy Maps." In *I3D '09: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, pp. 67–74. New York: ACM, 2009.

[Williams 78] Lance Williams. "Casting Curved Shadows on Curved Surfaces." *Computer Graphics (Proceedings of SIGGRAPH 78)* 12:3 (1978), 270–274.

[Xie et al. 07] Feng Xie, Eric Tabellion, and Andrew Pearce. "Soft Shadows by Ray Tracing Multilayer Transparent Shadow Maps." In *Rendering Techniques 2007: 18th Eurographics Workshop on Rendering*, pp. 265–276. Aire-la-Ville, Switzerland: Eurographics Association, 2007.

[Yang et al. 10] Jason Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. "Real-Time Concurrent Linked List Construction on the GPU." In *Rendering Techniques 2010: Eurographics Symposium on Rendering*, pp. 51–60. Aire-la-Ville, Switzerland: Eurographics Association, 2010.

[Yuksel and Keyser 08] Cem Yuksel and John Keyser. "Deep Opacity Maps." *Computer Graphics Forum* 27:2 (2008), 675–680.

4

IV

Fast Soft Shadows with Temporal Coherence

Daniel Scherzer, Michael Schwärzler
and Oliver Mattausch

4.1 Introduction

In computer graphics applications, soft shadows are usually generated using either a single shadow map together with some clever filtering method (which is fast, but inaccurate), or by calculating physically correct soft shadows with *light-source area sampling* [Heckbert and Herf 97]. Many shadow maps from random positions on the light source are created (which is slow) and the average of the resulting shadow tests is taken (see Figure 4.1).

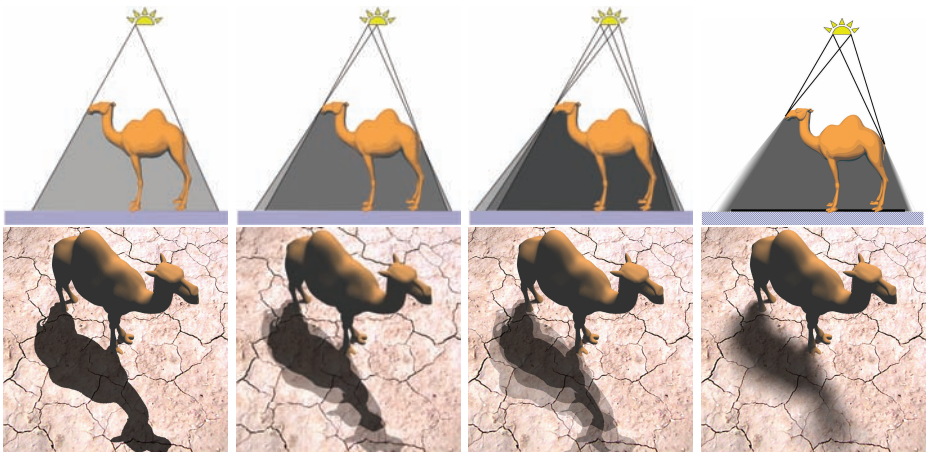


Figure 4.1. Light sampling with one, two, three and 256 shadow maps (*left* to *right*).

We present a soft shadow algorithm that combines the benefits of these two approaches by employing temporal coherence: the light source is sampled over multiple frames instead of a single frame, creating only a single shadow map with each frame. The individual shadow test results are stored in a screen-space (of the camera) *shadow buffer* (see Figure 4.2). Focusing each shadow map on creation can be done because only fragments in the screen space of the camera remain stored in the shadow buffer. This buffer is recreated with each frame using the shadow buffer from the previous frame B_{prev} as input (ping-pong style). The input B_{prev} holds shadowing information only for pixels that were visible in the previous frame. Pixels that become newly visible in the current frame due to camera (or object) movement (so-called disocclusions) have no shadowing information stored in this buffer. For these pixels we use spatial filtering to estimate the soft shadow results.

Our approach is faster as typical single sample soft shadow approaches like PCSS, but provides physically accurate results and does not suffer from typical single-sample artifacts. It also works on moving objects by marking them in the shadow map and falling back to a standard single-sample approach in these areas.

4.2 Algorithm

The main idea of our algorithm is to formulate light-source area sampling in an iterative manner, evaluating only a single shadow map per frame. We start by looking at the math for light-source area sampling: given n shadow maps, we can calculate the soft-shadow result for a given pixel $\mathbf{p}$ by averaging over the hard-shadow results s_i calculated for each shadow map. This is given by

$$\psi_n(\mathbf{p}) = \frac{1}{n} \sum_{i=1}^{n} s_i(\mathbf{p}). \tag{4.1}$$

We want to evaluate this formula iteratively by adding a new shadow map at each frame, combining its shadow information with the data from previous frames that have been stored in a so-called shadow buffer B_{prev}, and storing it in a new shadow buffer B_{cur}. With this approach, the approximated shadow in the buffer improves from frame to frame and converges to the true soft-shadow result.

Our approach has the following steps (see also Figure 4.2):

1. Create a shadow map SM from a randomly selected position on the area light.
2. Create a new shadow buffer B_{cur} with B_{prev} and SM as input. For each screen pixel we do the following steps:
   (a) Calculate the hard shadow result from SM (see Listing 4.1).

(b) Check if the pixel was visible in the last frame and therefore has associated shadowing information stored in the shadow buffer (see Section 4.2.1):

Yes: Combine information from the shadow buffer with SM (see Section 4.2.2).

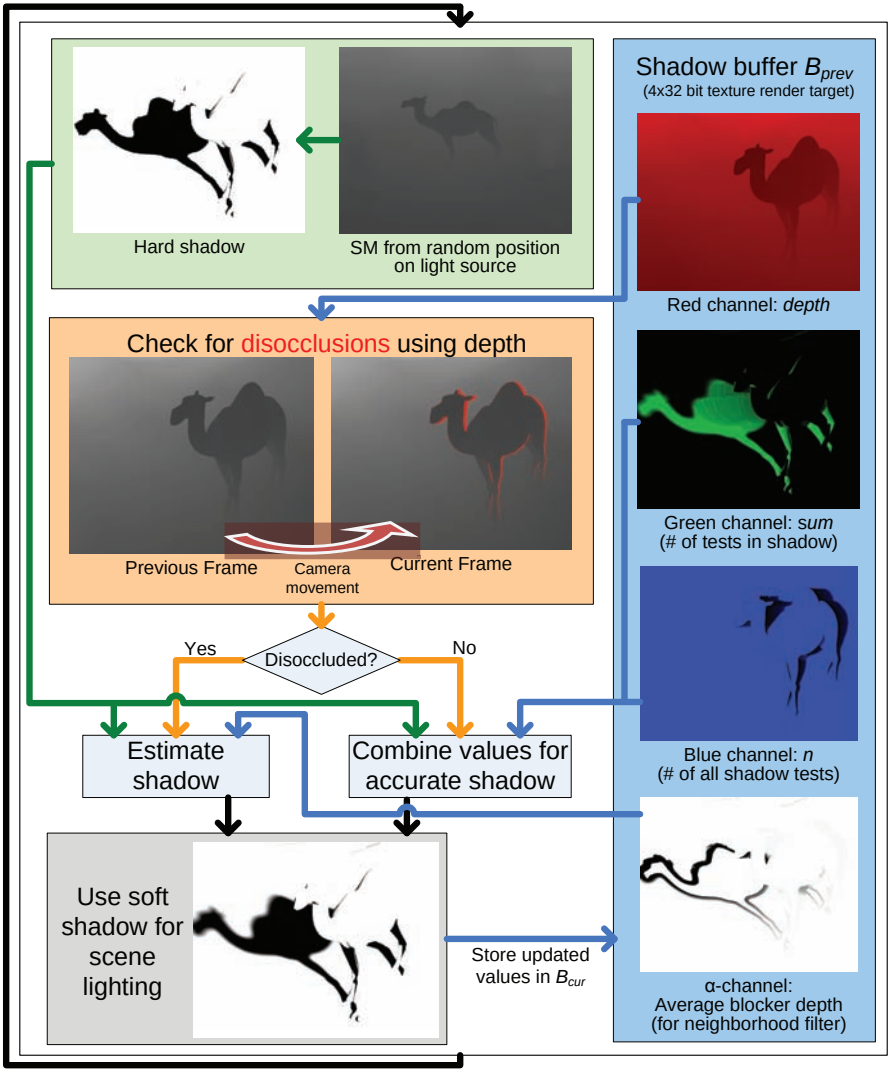


Figure 4.2. Structure of our algorithm.

No: Use a depth-aware spatial filtering approach that incorporates neighboring shadow buffer values to generate an initial soft shadow estimation for this pixel (see Section 4.2.3).

We will now describe the individual steps of this algorithm in more detail (including code fragments from our HLSL 4.0 pixel shader). The first step is straightforward: a random position on the light source is selected. This position is then used as a point light source from which a shadow map is created. For creating the shadow map and evaluating the hard shadow results any algorithm can be used, for instance *LiSPSM* [Wimmer et al. 04] or *silhouette shadow maps* [Sen et al. 03]. Then we start to create the new shadow buffer B_{cur} by using the shadow buffer from the previous frame B_{prev} and SM as input.

```
//shadow map sampling coordinates
const float2 smCoord = texSpace(input.LightPos);
//Linear depth of current pixel in light space
const float fragDepth = input.LightPos.z;
//sample depth in shadow map
const float Depth = getSMtexel(smCoord);
//store hard shadow test result as initial sum
float ShadowSum = shadowTest(Depth,fragDepth);
```

Listing 4.1. Hard shadow test.

4.2.1 Was This Pixel Visible?

We first have to check if the pixel $\mathbf{p}_{\text{cur}}$ we are looking at was visible in the previous frame and can therefore be found in B_{prev}. The process to determine this is called *temporal reprojection* [Scherzer et al. 07], We back-project it (thereby accounting for camera movement) into the coordinate system where B_{prev} was created. We transform $\mathbf{p}_{\text{cur}}$ therefore, from the post-perspective space of the current view back into the post-perspective space of the previous frame. Since we have the 3D position of our current pixel, we can simply use the view ($\mathbf{V}$) and projection ($\mathbf{P}$) matrices and their inverses of the current frame and the previous frame to do the transformation (see Figure 4.3):

$$\mathbf{p}_{prev} = \mathbf{P}_{prev} * \mathbf{V}_{prev} * \mathbf{V}_{cur}^{-1} * \mathbf{P}_{cur}^{-1} * \mathbf{p}_{cur}$$

To detect pixels that were not visible in the previous frame we first check if $\mathbf{p}_{\text{prev}}$ is inside B_{prev} in the x- and y-direction and then we check the z (i.e., the depth) difference between $\mathbf{p}_{\text{prev}}$ and the corresponding entry in B_{prev} at position $\mathbf{p}_{\text{prev}}$. If this difference exceeds a certain threshold, we conclude that this pixel was not visible in the previous frame (see Listing 4.2 and 4.3).

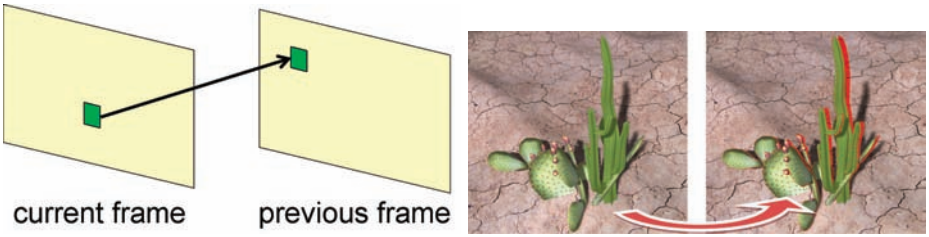


Figure 4.3. Back-projection of a single pixel (left). If we do this for every pixel we virtually transform the previous frame into the current, except for the pixels that were not visible in the previous frame (shown in red (right)).

```
bool outsideTexture(float2 Tex) {
  return any(bool2(Tex.x < 0.0f, Tex.y < 0.0f))
    || any(bool2(Tex.x > 1.0f, Tex.y > 1.0f));
}
```

Listing 4.2. Helper function for checking the validity of texture coordinates.

The obtained position will normally not be at an exact pixel center in B_{prev} except in the special case that no movement has occurred. Consequently, texture filtering should be applied during the lookup in the shadow buffer B_{prev}. In practice, the bilinear filtering offered by graphics hardware shows good results.

```
//previous shadow buffer sampling coordinates:
const float2 shadowBuffTexC = texSpace(input.BufferPos);
//check if the pixel is inside the previous shadow buffer:
if(!outsideTexture(shadowBuffTexC)) {
  //inside of previous data -> we can try to re-use information!
  float4 oldData = getShadowBufferTexel(shadowBuffTexC);
  const float oldDepth = oldData.x;
  //check if depths are alike, so we can re-use information
  if(abs(1-input.BufferPos.z/oldDepth) < EPSILON_DEPTH) {
    //old data available -> use it, see next section
    ...
  }
}
```

Listing 4.3. Test if the data for the current pixel was available in the previous shadow buffer.

4.2.2 Using the Shadow Buffer B_{prev}

In regions where no disocclusions occurred, B_{prev} holds shadowing information gathered over all the previous frames. Every new, additionally generated shadow map SM improves the accuracy of this soft-shadow information, and the current result has to be combined, therefore, with the already existing data.

SM allows us to calculate hard shadow results for the current frame, and together with the stored n and sum values in B_{prev}, the accurate shadow can easily be computed by (see Listing 4.4)

1. add the current shadow amount to the sum of all shadows,
2. increase the count n by 1,
3. divide the new sum by the new n.

```
//introduce better names for data from previous shadow buffer
const float oldSum = oldData.y;
const float oldCount = oldData.z;
//add shadow buffer sum to the current sum
ShadowSum += oldSum;
//increment n
float n = oldCount + 1;
//calculate soft shadow
float softShadow = ShadowSum / n;
```

Listing 4.4. Combination of a hard shadow and the data from the shadow buffer.

4.2.3 Soft Shadow Estimation in Disoccluded Areas

If the pixel is not in B_{prev}, we cannot calculate a soft shadow using Equation 4.1 as described in Section 4.2.2, since we have only one shadow-map test result of which to calculate the average. We therefore generate an initial soft-shadow estimation for this pixel by a applying a depth-aware spatial filter (bilateral filter) (see Listing 4.5), which takes neighboring pixels (distributed on a Poisson disk) in the shadow buffer B_{prev} into account if they lie on a similar depth.

```
float neighborhoodFilter(const float2 uv,
                         const float2 filterRadiusUV,
                         const float currentDepth) {
   float sampleSum = 0, numSamples = 0;
   for(int i = 0; i < NUM_POISSON_SAMPLES; ++i ) {
      const float2 offset = poissonDisk[i] * filterRadiusUV;
      const float3 data = getShadowBufferTexel(uv + offset).xyz;
```

```
        const float depth = data.x;
        const float sum = data.y;
        const float n = data.z;
        if(abs(1-currentDepth/depth) < EPSILON_DEPTH) {
            sampleSum += sum/n;
            numSamples++;
        }
    }
    return numSamples > 0 ? sampleSum/numSamples : -1;
}
```

Listing 4.5. Soft shadow estimation by filtering the shadow buffer neighborhood.

If these neighboring pixels have not been recently disoccluded, they are very likely to provide a good approximation of the correct soft-shadow value and will help to avoid discontinuities between the shadowed pixels.

The filter radius is calculated using the same penumbra estimation as in the PCSS algorithm [Fernando 05]. The idea is to approximate all occluders in a search area around the pixel by one planer occluder at depth z_{avg}. Using the intercept theorem and the relations between pixel depth z_{receiver} and light source size w_{light} an estimation of the penumbra width w_{penumbra} (see Figure 4.4) is given by

$$w_{\text{penumbra}} = w_{\text{light}} \frac{(z_{\text{receiver}} - z_{\text{avg}})}{z_{\text{avg}}}.$$

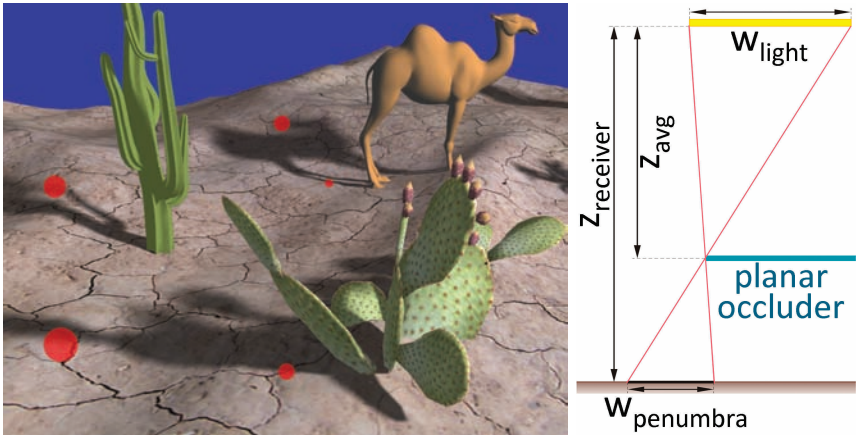


Figure 4.4. The sampling radius of the neighborhood filter depends on the scene depth and an estimated penumbra size (left). The penumbra width can be approximated by using the intercept theorem (right).

The calculation of the average occluder depth is done by searching for potential blockers in the shadow map, and is therefore a computationally costly step—but in contrast to PCSS, we have to do this step only in the case of a disocclusion. Otherwise, we store it in the shadow buffer for further use in consecutive frames (see Section 4.2.4).

In practice, it has been found useful to assign a weight larger than 1 to this approximation (for one hard shadow map evaluation), to avoid jittering artifacts in the first few frames after a disocclusion has occurred. Therefore, we use the number of Poisson samples from the neighborhood filter as weight.

4.2.4 Putting It All Together

In order to avoid visible discontinuities when switching from the estimate generated after a disocclusion and the correct result obtained from the shadow buffer B_{prev}, the two shadow values are blended. This blended shadow is only used to improve the visual quality in the first few frames and is not stored in the shadow buffer. Note that we do not have to estimate the average blocker depth for the neighborhood filter again, as it has been evaluated and stored in the shadow buffer directly after the disocclusion! Additionally, this average blocker depth is refined every frame by adding the additional depth value from the current shadow map SM (see Listing 4.6).

```
//load average blocker depth from the previous frame
const float oldAvgBlockerDepth = oldData.w;
//if first frame after dissoclusion only one depth is available
if(1.0 == ShadowSum) avgBlockerDepth = Depth;
//Update average blocker depth
if(oldAvgBlockerDepth >= 0.0f) {
  if(avgBlockerDepth >= 0.0f) {
    float sum = oldAvgBlockerDepth*(n-1);
    sum += avgBlockerDepth;
    avgBlockerDepth = sum/(n);
  }
  else
    avgBlockerDepth = oldAvgBlockerDepth;
}
```

Listing 4.6. Iterative refinement of the average blocker depth

To derive a formula for the blending weight we use a statistical approach: we estimate the standard error s of our sampling-based soft-shadow solution with a

binomial variance estimator

$$s_n(\mathbf{p}) = \sqrt{\frac{\psi_n(\mathbf{p})(1-\psi_n(\mathbf{p}))}{n-1}}$$

This formula allows us to estimate the quality of our soft-shadow solution after taking n shadow maps (for details see [Scherzer et al. 09]). If this standard error is above a certain user-defined threshold $err_{\max}$, we use only the approach described in Section 4.2.3. If the standard error is below a certain second threshold $\mathrm{err}_{\min}$, we use only the sampling-based soft shadow. Between these bounds, we blend the two soft shadow results.

```
//calculate standard error with binomial variance estimator
const float error =
  n == 1.0 ? 1.0 : sqrt(softShadow*(1-softShadow)/(n-1));
//if we have recently disoccluded samples or a large error,
//support the shadow information with an approximation
if(error >= err_min && avgBlockerDepth > 0) {
  //penumbra estimation like in PCSS, but with the average
  //occluder depth from the history buffer
  const float penumbraEstimation = vLightDimensions[0] *
    ((fragDepth - avgBlockerDepth) / avgBlockerDepth);
  //do spatial filtering in the shadow buffer (screen space):
  const float depthFactor = (nearPlaneDist/input.Depth);
  const float shadowEstimate = neighborhoodFilter(
    shadowBuffTexC, vAspectRatio*depthFactor*penumbraEstimation,
    input.Depth);
  //if shadow estimate valid calculate new soft shadow
  if(shadowEstimate > 0.0f) {
    if(inDisoccludedRegion) {
      //disoccluded sample: only estimated shadow
      //define weight for estimate
      const float estimateWeight = NUM_POISSON_SAMPLES;
      ShadowSum = shadowEstimate*estimateWeight;
      n = estimateWeight;
      softShadow = shadowEstimate;
    } else {
      //blend estimated shadow with accumulated shadow
      //using the error as blending weight
      const float weight = (err_max-error)/(err_max-err_min);
      softShadow =
        shadowEstimate * (1-weight) + softShadow * weight;
    }
  }
}
```

Listing 4.7. The standard error decides which method is used.

After having the soft-shadow result evaluated for each displayed pixel, the final steps are to

- use the calculated result to modify the scene illumination, and output the shadowed scene on the corresponding render target, and to
- store the current depth, the number of successful shadow tests sum, the number of samples n, and the average blocker depth in the new shadow buffer render target B_{cur}.

```
output.Col1 = float4(input.Depth,ShadowSum,n,avgBlockerDepth);
//output the illuminated and shadowed image
output.Col0.rgb = LightingFunction(..) * (1-softShadow);
return output;
```

Listing 4.8. Store values in shadow buffer and output rendered image.

4.2.5 Moving Objects

Special care must be taken when it is necessary to handle moving objects, since they frequently produce disocclusions. Moreover, only the most recently accumulated shadow tests in the shadow buffer provide valid shadow information: as these shadow-casting objects move on, older shadow tests originate from different spatial scene compositions, and reusing them would lead to strong streaking artifacts.

We therefore identify shadows that are cast by moving objects by storing their depth in the shadow map SM with a negative sign (generating no additional memory cost) and checking whether this value is negative during the lookup in the shadow map. If this is the case (i.e., if we have a shadow that is cast by a moving object), we reduce the corresponding weight by setting sum and n to a low value. This allows new shadow tests to have more impact on the result, and removes the streaking artifacts at the cost of the physical correctness of the shadow. Please note that an example shader, in which moving objects are properly handled, is included in the accompanying demo application.

4.3 Comparison and Results

The proposed soft-shadowing algorithm offers a way to render physically accurate, high quality soft shadows in real-time applications at the same speed as today's fastest single-sample methods. Figure 4.5 shows a benchmark that compares our method to a fast PCSS version using only 16 texture lookups for the blocker search and 16 texture lookups for the PCF kernel on a 720p viewport.

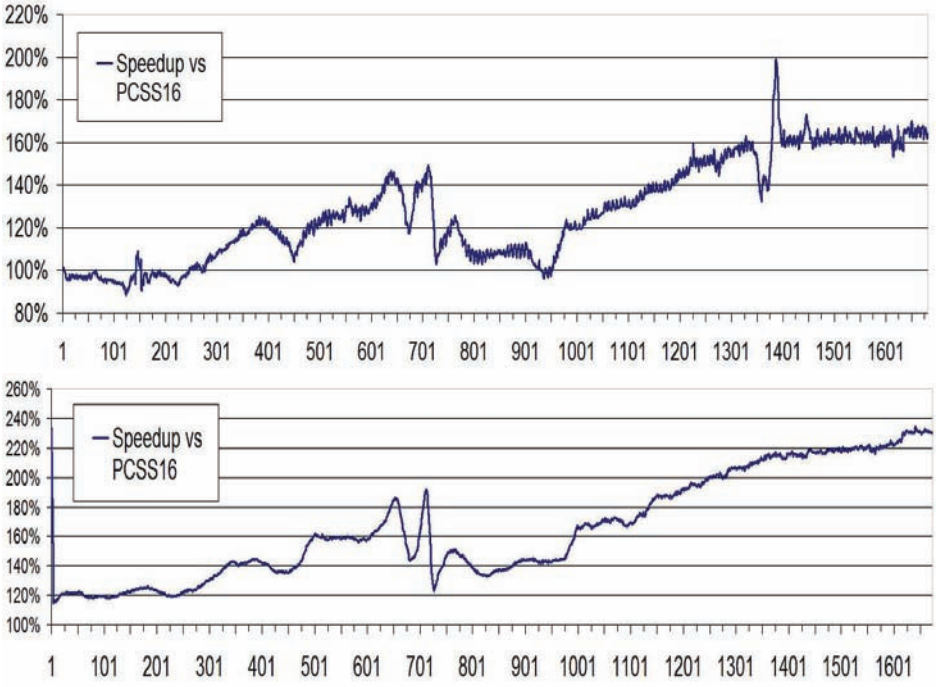


Figure 4.5. Speedup of our new algorithm in comparison to PCSS16 on a GeForce 280GTX (top) and a 9600MGT (bottom).

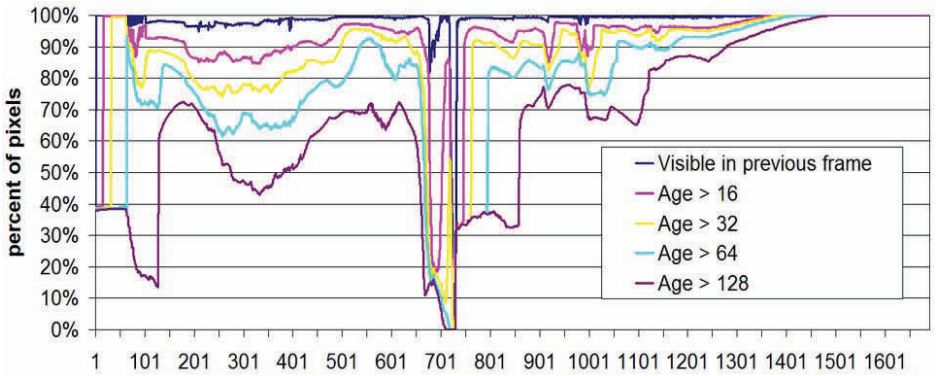


Figure 4.6. The "age" (i.e., the number of reusable shadow tests) of the fragments in our walkthrough scene.

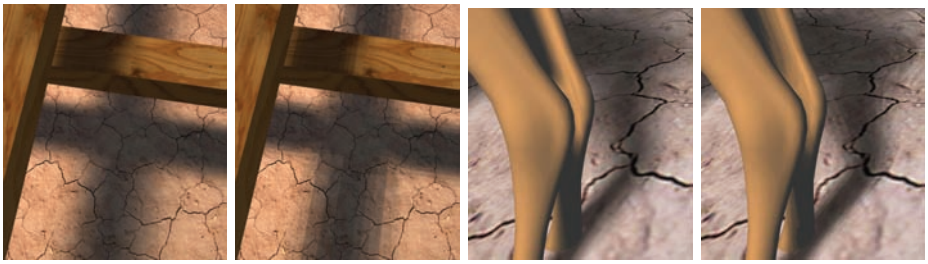


Figure 4.7. Overlapping occluders (our method, PCSS 16/16) and bands in big penumbrae (our method, PCSS 16/16) are known problem cases for single sample approaches left to right:.

Our algorithm tends to have a slower frame rate in cases of numerous disocclusions, because it has to perform the additional blocker search for the penumbra estimation. Due to its higher complexity (more ifs), our shader can be slower than PCSS in such circumstances. As soon as the shadow buffer is exploited and its values can be reused, our approach can unfold its strength and deliver higher frame rates, while PCSS still has to do the shadow map lookups. As can be seen in Figure 4.6, the number of fragments for which buffer data can be reused is usually high enough to obtain frame rates exceedings those that can be obtained with PCSS.

In static scenes, the soft shadows generated with our method are physically accurate and of a significantly better quality than is produced by PCSS, which suffers from typical single-sample artifacts (see Figure 4.7). For moving objects, the shadow buffer can hardly be exploited, and we therefore provide a fallback solution in which spatial neighborhood filtering is applied. Though humans can hardly perceive the physical incorrectness in such cases, there is room for improvement, since some flickering artifacts may remain when dynamic shadows overlap with static shadows that have large penumbrae.

Bibliography

[Fernando 05] Randima Fernando. "Percentage-Closer Soft Shadows." In *ACM SIGGRAPH Sketches*, p. 35. New York: ACM, 2005.

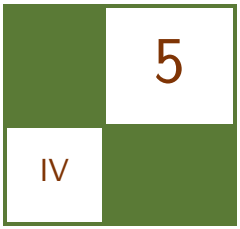
[Heckbert and Herf 97] Paul S. Heckbert and Michael Herf. "Simulating Soft Shadows with Graphics Hardware." Technical Report CMU-CS-97-104, CS Dept., Carnegie Mellon University, 1997.

[Scherzer et al. 07] Daniel Scherzer, Stefan Jeschke, and Michael Wimmer. "Pixel-Correct Shadow Maps with Temporal Reprojection and Shadow Test Confidence." In *Proceedings Eurographics Symposium on Rendering*, pp. 45–50. Aire-la-Ville, Switzerland: Eurographics Association, 2007.

[Scherzer et al. 09] Daniel Scherzer, Michael Schwärzler, Oliver Mattausch, and Michael Wimmer. "Real-Time Soft Shadows Using Temporal Coherence." In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II*, *Lecture Notes in Computer Science (LNCS)*, pp. 13–24. Berlin, Heidelberg: Springer-Verlag, 2009.

[Sen et al. 03] Pradeep Sen, Mike Cammarano, and Pat Hanrahan. "Shadow Silhouette Maps." *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 22:3 (2003), 521–526.

[Wimmer et al. 04] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. "Light Space Perspective Shadow Maps." In *Proceedings of Eurographics Symposium on Rendering 2004*. Aire-la-Ville, Switzerland: Eurographics Association, 2004.



Mipmapped Screen-Space Soft Shadows

Alberto Aguado and Eugenia Montiel

This chapter presents a technique for generating soft shadows based on shadow maps and screen space image filtering. The main idea is to use a mipmap to represent multifrequency shadows in screen space. The mipmap has two channels: the first channel stores the shadow intensity values and the second channel stores screen-space penumbrae widths. Shadow values are obtained by filtering while penumbrae widths are propagated by flood filling. After the mipmap is generated, the penumbrae values are used as indices to the mipmap levels. Thus, we transform the problem of shadow generation into the problem of selecting levels in a mipmap. This approach is extended by including layered shadow maps to improve shadows with multiple occlusions.

As with the standard shadow-map technique, the computations in the technique presented in this chapter are almost independent of the complexity of the scene. The use of the shadow's levels of detail in screen space and flood filling make this approach computationally attractive for real-time applications. The overhead computation compared to the standard shadow map is about 0.3 ms per shadow map on a GeForce 8800GTX.

5.1 Introduction and Previous Work

The shadow map technique is a well-known method for generating real-time shadows [Williams 78]. It is widely used in applications since it is computationally attractive and it is capable of dealing with complex geometry. The original technique determines if a point is in a shadow by comparing the distances to the light and to the camera in two steps. First, the scene is rendered from the camera view point and the distance of the closest point is stored in the texture defining the shadow map. In the second step, the position of a point is transformed

into the camera frame, so its distance to the camera can be compared to the shadow-map value. This comparison determines if a point is occluded or not; thus points are either fully shadowed or fully illuminated. The binary nature of the comparison produces hard shadows, reducing the realism of the scene. As such, previous works have extended the shadow-map technique to produce soft shadows.

The technique presented in this chapter filters the result of the shadow map test. This approach was introduced in the percentage closer filtering (PCF) [Reeves et al. 87] technique. PCF determines the shadow value of a pixel by projecting its area into the shadow map. The shadow intensity is defined by the number of values in the shadow map that are lower than the value at the center of the projected area. Percentage closer soft shadows (PCSS) [Fernando 05] extended the PCF technique to include shadows of different widths by replacing the pixel's area with a sampling region whose area depends on the distance between the occluder and the receiver.

The PCSS technique is fast and it provides perceptually accurate soft shadows, so it has become one of the most often used methods in real-time applications. However, it has two main issues. First, since it requires sampling a region per pixel, it can require an excessive number of computations for large penumbrae. The number of computations can be reduced by using stochastic sampling, but it requires careful selection of the sampling region in order to avoid artifacts. Second, to determine the region's size, it is necessary to estimate an area in which to search for the blockers. In general, it is difficult to set an optimal size, since large regions lead to many computations and small regions reduce the shadows far away from the umbra.

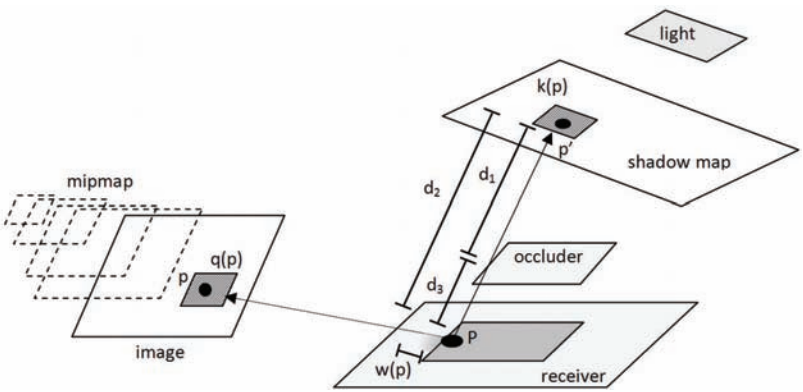


Figure 5.1. The mipmap in image space is built by using the result of the shadow map test and the distances between the occluder and the receiver.

In [Gambau et al. 10], instead of filtering by sampling the shadow map, soft shadows are obtained by filtering the result of the shadow-map comparison in screen space. The technique in this chapter follows this approach and it introduces a mipmap to represent multiple-frequency shadow details per pixel. As such, the problem of filtering is solved by selecting a mipmap level for each pixel. Filtering via mipmapping has been used in previous shadow-map techniques such as convolution shadow maps [Annen et al. 08] and variance shadow maps [Donnelly and Lauritzen 06]. In the technique presented in this chapter, mipmapping is used to filter screen-space soft shadows. In addition to storing multi-frequency shadows, the mipmap is exploited to propagate occlusion information obtained from the occluder and the shadowed surface. Occlusion information is used to select the shadow frequency as an index to a level in the mipmap. The efficiency of the mipmap filtering and the screen-space computations make it possible to create penumbrae covering large areas, expending little computational overhead, and performing no stratified sampling.

5.2 Penumbra Width

Figure 5.1 illustrates the geometry of the scene, the camera, and the shadow map. A point P in a surface is mapped to the point p in the image and to the point p' in the shadow map by the inverse of the camera and light transformations, respectively. The shadow map stores the closest distance to any surface from the light viewpoint. This is the distance denoted as d_1 in Figure 5.1. Thus, p will be in shadow when the distance in the shadow map is lower than the distance to the point P. That is, the shadow intensity of the pixel p is zero if $d_1 < \quad d_2$, and 1 otherwise.

Soft shadows can be generated by filtering the result of the shadow-map comparison in a region $k(p)$ in the shadow map. Using PCF, the region is defined by the projection of the area of the pixel p into the shadow map. With PCSS, the region is determined based on the estimation of the penumbrae sizes and it is implemented in two steps. The first step searches for the occluder of the point P by averaging the values in the shadow map close to p'. In general, there is no way to determine the size of the search region and it is set as a parameter according to the scene. Once the distance to the occluder has been determined, the second step estimates the penumbra width by considering the geometry of the occluder, the receiver, and the light. This is defined as

$$w(p) \;=\; \frac{d_2\,(p) - d_1\,(p)}{d_1\,(p)} L\ .$$

Here, the value of L represents the size of the light and it is used to control the penumbrae; by increasing L, shadows in the scene become softer. Once the penumbra width is computed, the shadow can be filtered by considering a region

$k(p)$ whose size is proportional to it. That is, PCSS uses a proportional constant to map the penumbra width to the shadow map region $k(p)$.

The technique presented in this chapter is based on the PCSS penumbra estimation, but the values are computed only for the points that pass the shadow-map test. This is because, for these points, it is not necessary to compute averages to obtain the occluders' distances; they can be obtained by fetching the value for the point p' in the shadow map. That is, d_1 is simply defined by the shadow map entry. Another difference from PCSS is that the penumbra estimation is not used to define a filter of the values in the shadow map, but it is used to define a filter in image space.

5.3 Screen-Space Filter

An estimate of the penumbra width in image space can be obtained by projecting $w(p)$ into the image plane. That is,

$$q(p) = \frac{f}{z} w(p).$$

Here, f is the focal length and z is the depth of the fragment. Thus, f/z accounts for the perspective scale. The focal length is a constant parameter that defines the distance between the center of projection and the image plane. In general, the area will be dependent also on the orientation between the camera and the surface. However, since we use isotropic filters, we cannot transform the area using particular directions.

The computation of $q(p)$ can be performed during the shadow map test as shown in Listing 5.1. In this implementation, the variable "pos" stores the position of the fragment in the world space, and it is used to compute the position in light space. The world-space and light-space positions are then used in the shadow-map test and in the computation of the penumbra width. The result of the shadow map is stored in the red channel of the render target and the penumbra width is stored in the blue channel. Notice that the implementation does not compute occlusion values for all the pixels, but some will have the $0\times$FFFFFFFF value.

```
void main(INPUT input, inout OUTPUT output)
{
    // Position in light space
    float4 light_pos = mul (pos, light_matrix)

    // Shadow map value
    float shadow_map_val = tex2D(shadowmap_texture0, light_pos);

    // Shadow map test
    output.color.r = shadow_map_val < light_pos.z - shadow_bias;
```

```
    // Penumbra width
    output.color.b = 0xFFFFFFFF;
    if(output.color.r == 1.0f) {
            float distance_factor =
               (light_pos.z / shadow_map_val) - 1.0f;
            output.color.b = distance_factor * L * f * A / pos.z;
    }

    // Tag the region for region filling
    output.color.g = output.color.r
}
```

Listing 5.1. Performing the shadow test and computing the penumbrae widths in two different texture channels.

Figure 5.2 shows examples of the penumbrae values. The image on the left shows an example scene. The image in the middle shows the penumbrae widths. The white pixels are points without penumbrae widths. That is, they are points without occlusions. The pixels' intensities represent penumbrae widths and they show the dependency between the occluder and the receiver positions. Occluders that are far away from the receiver have lighter values than occluders that are closer. Lighter values indicate large smooth shadows while dark values indicate that shadows should have well-delineated borders. The image on the right shows the result of the shadow-map test computed in Listing 1. The pixels in this image have just two intensities that define hard shadows.

Notice that the code in Listing 5.1 computes penumbrae estimations only for pixels in the hard shadow region. However, in order to compute soft shadows, it is necessary to obtain estimates of the points that will define the umbra of the shadow. The penumbrae widths for these pixels can be obtained by searching for the closer occluder. Here, the search is performed by a flood-filling technique implemented in a mipmap. In this approach, each level of the mipmap is manually

Figure 5.2. Example scene (left). Penumbra widths, light values indicate large penumbrae (middle). Hard shadows (right).

created by rendering a single quad. The pixel shader sets as render target the mipmap level we are computing, and it takes as resource texture the previous mipmap level. The advantage of this approach is that the reduction in resolution at each level causes an increase in the search region. Thus, large regions can be searched with a small number of computations.

The flood-filling implementation requires distinguishing between pixels that have been filled and pixels that need to be filled. This can be efficiently implemented by using a single bit in a texture. To simplify the presentation, the implementations in Listing 5.1 and Listing 5.2 use an extra channel on the texture. In Listing 5.1 the pixels that define the filled region are tagged by setting the green channel to one. This channel is used, when the mipmap level is created, as shown in Listing 5.2 to distinguish pixels that should be used during the flood fill. In Listing 5.2, the value of a pixel is obtained by averaging the values that form the fill region using a 5×5 window in the source texture. The implementation averages distances so pixels so that are close to several occluders do not produce sharp shadow discontinuities.

```
void main(INPUT input, inout OUTPUT output)
{
    // The sum of all values in the region
    float sum = 0.0f;

    // Number of points in the region
    float num = 0.0f;

    // Flood fill using a window 5x5
    for (int i = 0; i<5; i++) {
        for (int j = 0; j<5; j++) {
            float4 t = float4( uv.x - target_shift.x
                                * (-2.5f + i),
                                uv.y - target_shift.y
                                * (-2.5f + j),
                                0, previous_level);

            // Read input level
            float4 val = tex2Dlod(samMipMap, t);

            // Flood fill averaging region pixels only
            if(val.g == 1.0f){
                sum += val.b;
                num++;
            }

        }
    }

    // Output flood fill value
    if( num>0.0f ) {
```

```
            // Pixel should be flood
            output.color.b = sum / num;
            output.color.g = 1.0f;
        }
        else{
            // Pixel is not in the flood region
            output.color.b = 0xFFFFFFFF;
            output.color.g = 0.0f;
        }
    }
```

Listing 5.2. The mipmap level generation. Flood filling for occlusion values.

Figure 5.3 shows an example of the results of the flood-filling implementation. At each level the penumbrae widths are propagated, covering large regions in the image. Since each level has half-resolution, a pixel covers four times the area of the previous level. Thus, large regions are covered using few levels.

Figure 5.3. Mipmap levels obtained by flood filling. The flood fill propagates information from hard shadows to outer regions.

5.4 Filtering Shadows

The penumbrae widths in the mipmap define the filtering that should be applied to the hard shadows in order to obtain soft shadows. Although it is possible to apply a filter for each pixel, a more efficient implementation can be obtained by using a multi-frequency representation. The main advantage is that frequency filtering can be used to reduce the data, so filters of large regions can be performed with small kernels applied to few pixels. That is, large filters correspond to small kernels in low-resolution images.

```
void main(INPUT input, inout OUTPUT output)
{
    // The sum of all values in the region and number of points
    float sum = 0.0f;
    float num = 0.0f;

    // Stores result of filter
    output.color.b = 0.0f;
```

```
    // Evaluate using a window 5x5
    for (int i = 0; i<5; i++) {
        for (int j = 0; j<5; j++) {
            float4 t = float4(uv.x - target_shift.x
                        * (-2.5f + i),
                        uv.y - target_shift.y * (-2.5f + j),
                        0, previous_level);
            // Read input level
            float4 val = tex2Dlod(samMipMap, t);

            // Flood fill averaging region pixels only
            if(val.g == 1.0f){
                sum += val.b;
                num++;
            }
            // Gaussian filter
            output.color.b += val.r * kernel[i][j];
        }
    }
    // Output flood fill value
    if( num>0.0f ) {
        output.color.b = sum / num;
        output.color.g = 1.0f;
    }
    else{
        output.color.b = 0xFFFFFFFF;
        output.color.g = 0.0f;
    }
}
```

Listing 5.3. Mipmap level generation including the filter of the shadow map test values. The red channel stores the shadow intensity, the blue channel stores the penumbra width and the green channel is used to tag the filled region.

When creating a multi-frequency representation, it is important to select the cutoff frequencies to avoid aliasing [Bracewell 00]. Since the resolution of the mipmap is reduced by half at each level, aliasing is avoided by reducing the frequency by half. This cutoff frequency is obtained by a Gaussian filter with standard deviation set to one. In the implementation, the filter is computed at the same time as the flood-filling in Listing 5.2, but on a different texture channel. Listing 5.3 includes the filtering to the mipmap level generation shader. The kernel matrix defines the normalized coefficients of the Gaussian filter.

Figure 5.4 shows an example of the results obtained when applying the filtering. The top row shows the mipmap levels. Notice that high resolutions give fine detail with well-delineated shadow borders while coarse resolutions have soft borders. The images in the bottom row were obtained by using the shadows defined on the corresponding image on the top row. That is, the same filter size is used for the whole image, so penumbrae have the same widths. This example shows that

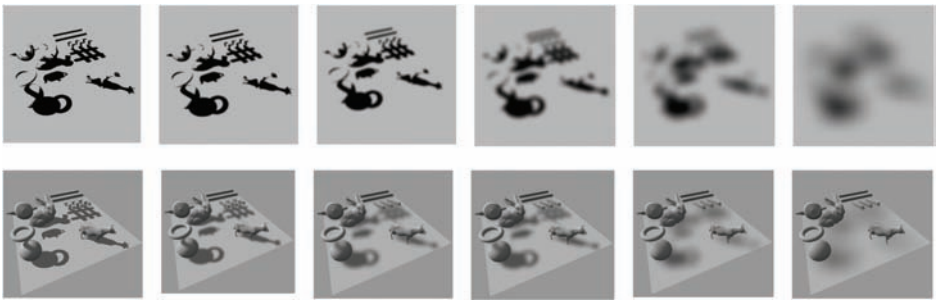


Figure 5.4. Mipmap levels obtained by filtering the result of the shadow map test (top). Shadows defined at each mipmap level (bottom).

image-space filters can be used to generate shadows with different penumbrae; however, to obtain compelling shadows in the image, we should select different frequencies for different pixels. That is, we have changed the problem of shadow generation into a problem of selecting levels in a mipmap.

5.5 Mipmap Level Selection

The flood-fill and filtering processes create a mipmap with two channels: the first channel contains shadows filtered with different cutoff frequencies and the second channel contains penumbrae widths. Small penumbrae widths values indicate close contact points or close lights, so the shadows should be well delineated. That is, the pixel should be rendered using the shadows on the first levels of the mipmap. Large values indicate that the distance between the occluder and the receiver is significant with respect to the distance and size of the light. Thus, pixels should use the shadows in the low-resolution levels; the penumbrae width determines the frequency content in each pixel.

In order to select the mipmap level for each pixel, we relate the penumbrae widths to filter sizes by considering the fact that since filters are applied at each resolution, the size of the filter is squared on each level. That is, the width of the filter at level i is

$$s = 2^i.$$

Thus, if we consider that the width of the penumbra is defined by the size of the filter, then a penumbra width will be generated at the level

$$i = \log_2 \left(q \left(p \right) \right).$$

Notice that this equation does not give integer levels; we should not be limited by the values in the mipmap levels, and we can generate shadows for intermediate values. In the implementation, we use bilinear interpolation to compute the

shadows between mipmap levels. This generates a variation of shadows and produces smooth transitions in the penumbrae.

```
void main(INPUT input, inout OUTPUT output)
{
    // Init shadow intensity
    float shadow = 1.0f;

    // Fetch mipmap levels
    float4 val[8];
    for( int level =0; level < 8; level++ ){
        val[level]= tex2Dlod(g_samMipMap, float4(uv,0,level);
    }

    // Find q(p)
    float q = 0;
    for( int level = 0; level < MAX_LEVELS {\&}{\&} q == 0 ;
      level++ ){
        if (val[level].y != 0) q = val[level].z;
    }

    if( q>0.0f )
    {
        // Selected level
        if(q<1) q = 1;
        float l = log2(q);
        if(l > MAX_LEVELS) l = MAX_LEVELS - 0.1f;

        // Interpolate levels
        int down = floor(l);
        int up = down + 1;
        float interp = l-down;

        // Shadow intensity
        shadow = (1.0f - lerp(val[down].x,val[up].x,interp));
    }
output.color = shadow;
}
```

Listing 5.4. Shadow intensity computation from the mipmap.

The selection of levels implementation is outlined in Listing 5.4. The code starts by fetching all the levels of the mipmap. The fetching step uses interpolation, so we obtain smooth shadow values. The value of $q(p)$ is obtained by looking for the first penumbra value in the mipmap levels. This value is then used to compute the shadow intensity by interpolating a pair of selected levels.

The shadow intensity should be added to the final rendering of the scene. The way this is performed depends on the rendering type. Figure 5.5 illustrates

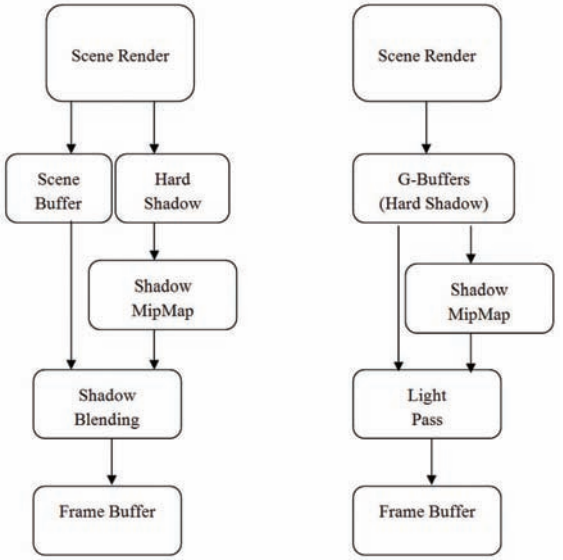


Figure 5.5. Mipmapped shadows implemented using the main scene rendering. Forward rendering (left). Deferred rendering (right).

how shadows can be added to the scene in forward and deferred rendering. In forward rendering, as shown in Figure 5.5 (left), hard shadows are computed during the main rendering of the scene, and they are stored in the bottom level of the mipmap. Thus, the scene buffer does not contain any shadows. Afterward the mipmap is constructed and subsequently the scene is shadowed by a shadow blending post-processing. The post-processing computes the shadow intensities and it combines the scene buffer and the shadow intensity by rendering a single quad. In the deferred rendering illustrated in Figure 5.5 (right), hard shadows are stored in a G-buffer and the shadow map can be used during the lighting pass.

Figure 5.6 shows some examples of soft shadows generated by using the mipmap technique. The first two images were obtained by changing the light's area, so shadows become smoother. The third image shows a close-up view to highlight contact shadows. As with any other shadow-map techniques, accurate shadows at contact point and self-shadows require an appropriate bias in the shadow map test. The first image in the bottom row shows how shadows change depending on the distance between the occluder and the receiver. Shadows are blurred and smooth for distant points whilst they are well delineated close to contact points. The final two images show the result on textured surfaces.

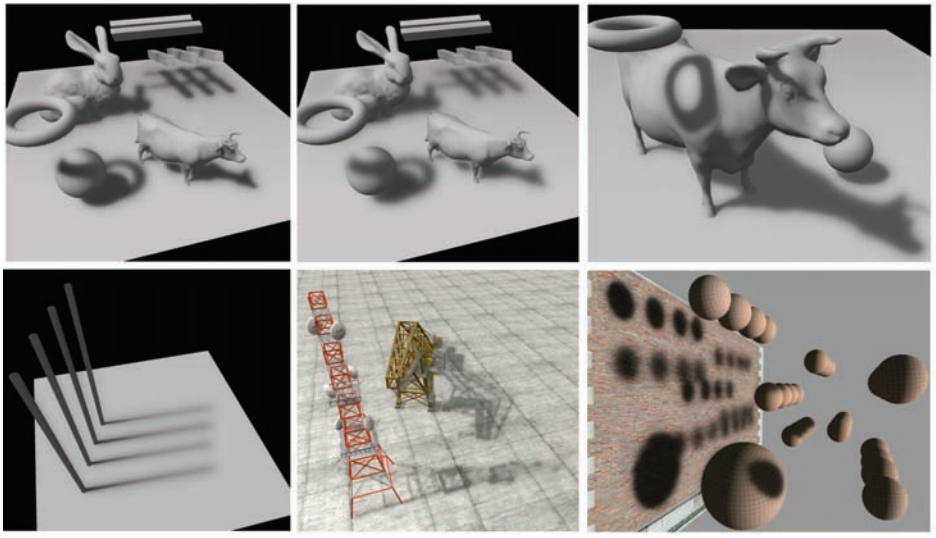


Figure 5.6. Examples of soft shadows.

5.6 Multiple Occlusions

In general, shadow-map techniques can suffer from light bleeding. This is because the shadow map stores a single value that represents the distance to the closest surface. Thus, when a surface has multiple occluders, the distance to the closest occluder cannot be computed. This is illustrated in Figure 5.7, which highlights two points on the receiver; one point has one occluder and the other has two occluders. For the dark point, the distance to the occluder is determined by subtracting the distance to the light from the distance in the shadow map (i.e., $d_2 - d_1$). This gives a good estimate of the occluder's distance. For the lighter point, the shadow map stores the distance to the closest occluder (i.e., d'_1). Thus, the occluder's distance is computed as $d_2 - d'_1$ and, consequently, the shadow will be overestimated. In some cases, as shown in Figure 5.6, the variations of shadow intensity may not produce notable changes in intensities. However, in several cases the difference between shadows can be very noticeable. This problem is more significant for semitransparent objects.

A straightforward approach to computing a better estimate of the distances between the receiver and the occluders is to perform a ray-tracing search; however, this will impose important computational constraints. An alternative approach is to store several values in the shadow map, so we can search for the closest occluder. A simple implementation of this approach can be developed using layered shadow maps.

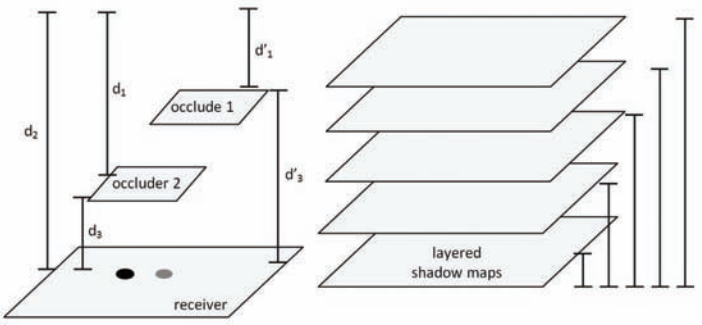


Figure 5.7. Light bleeding caused by multiple occlusion.

Layered shadow maps define an array of shadow maps that store the distances to the closest point for slices parallel to the light. In our implementation, each shadow map is obtained by rendering the scene multiple times, changing the near clip of the camera to cover the regions illustrated in Figure 5.7 (right). That is, the first shadow map covers a small region far from the light and the next shadow maps cover regions that increase in size approaching the camera.

In order to use layered shadow maps, the shadow-map test in Listing 5.1 should be changed to search for occluders in an array of textures. This is implemented in Listing 5.5. Here, the test is performed on each of the layer textures and the first layer that passes the shadow-map test is used to compute the occluder distance. It is important to mention that if multiple occluders are close to each other, then they will be located in the same shadow map. However, if they are close, then the distance error is low and the shadows are similar. That is, layers will not guarantee the correct distance computation, but they will mitigate problems caused by multiple occluders at far distances.

Figure 5.8 shows two examples that compare renderings with and without layered shadow maps. The images in the top row were obtained with a single shadow map while the images on the bottom row have eight layers. In the example shown in the images on the left, there are two multiple occlusions caused by the brick blocks and containers. Since the distance between the bricks and the container on the left is small relative to their distance to the light, shadows caused by both objects merge without causing artifacts. However, the large distance between the bricks and the container at the right causes a light shadow under the container. As shown in the image on the bottom row, these problems are reduced by using layered shadow maps.

```
void main(INPUT input, inout OUTPUT output)
{
```

```
    float4 light_pos = mul (pos4,light_matrix)

    // For each layer
    float2 shadow_map_val[8]
    shadow_map_val[0] = GetShadowMap(0,light_pos); // :
    shadow_map_val[7] = GetShadowMap(7,light_pos);

    // Shadow map test. Look for first occluder
    output.color.r = 0.0f;
    float distance = 0.0f;

    for( index =0; index<7;index++){
        if(shadow_map_val[index].x < light_pos.z - shadow_bias) {
            output.color.r = shadow_map_val[index].y;
            distance = shadow_map_value[index].x;
            index = 8;
        }
    }

    float distance_factor =
                        (light_pos.z / distance)--1.0f;
    output.color.b = distance_factor * L * f * A / pos.z;
}
```

Listing 5.5. Performing the shadow test and computing the penumbrae widths for layered shadow maps and transparency.

The example in the top-right image in Figure 5.8 shows light bleeding caused by a semitransparent object. Shadows for semitransparent objects can be created by changing the intensity of the shadows according to the transparency value of the albedo texture of the occluder. This modification can be implemented during the shadow-map test, so it does not add any significant computational overhead; it requires changing the shadow-map generation and the shadow-map test. The shadow-map creation should be modified so that the shadow map keeps the distance to the closest object and the alpha value of the albedo texture. The alpha value can then be used as shown in Listing 5.5 to determine the intensity of the shadow.

The computation of transparency, using values in the shadow map, is computationally attractive; however it can produce light bleeding for multiple occluders. This is illustrated in Figure 5.8 (top right). Here, the bin is causing a multiple occlusion with the bus stop glass. Thus, the shadow cast by the bin uses the transparency of glass and produces a very weak shadow. This is because a single shadow map stores only the alpha of the closest object to the light. As shown in Figure 5.8 (bottom right), layered shadow maps can alleviate this problem. However, if the occluders are moved close to each other, the layer strategy may fail to store multiple values and objects can produce incorrect shadows.

Figure 5.8. Examples of layered shadows. Light bleeding caused by incorrect computation of the distance between the occluder and the receiver (top). Layered shadow maps can reduce occluder problems (bottom).

5.7 Discussion

Compared with standard shadow maps, the technique presented in this chapter uses an extra texture to store the mipmap and one texture for each layer for the layered version. In terms of processing, it adds a computational overhead caused by: (i) the computations of the penumbra width during the shadow-map test (Listing 5.1); (ii) the creation of the mipmap (Listing 5.3); and (iii) the mipmap lookup during shadow blending (Listing 5.4).

The rendering times of the technique are shown in Table 5.1. The columns in the table show the frames per second when: rendering without shadows, rendering using standard shadow maps, rendering using the mipmap, and when using layers. The results were obtained for a test scene with 13K faces and by using a GeForce 8800GTX with a 720×480 display resolution. The implementation used a 512×512 shadow map and six mipmap levels. The frame time increases about 0.3 ms when the mipmap is used to generate soft shadows. This increase is mainly because of the time spent during the generation of the mipmap. In

	No Shadows	**Standard Shadow Map** (hard shadows)	**Mipmap Shadows**	**Layered Shadow Maps**
Frame Rate	345	340	305	230
Frame Time	2.89ms	2.94ms	3.27ms	4.34ms

Table 5.1. Frame rate for different implementations.

the layered version, the increase in rendering time is mainly due the multiple rendering required to create the shadow map for each layer. The time shown in Table 5.1 was obtained by considering six layers.

The computational load is adequate for real-time applications and the results show compelling smooth shadows. However, multiple occlusions can produce light bleeding. This is more evident as the light's area increases, since shadows with significantly different intensities can be created. This problem can be mitigated by saturating the intensity of the shadows or by using layered shadow maps. Nevertheless, when dealing with complex scenes and large-area lights, there still may be variations of intensities on multiple occlusion zones. As such, the technique could benefit from more elaborate layer placements or peeling layer strategies. Finally, it is important to mention that this technique relies on shadow maps, so it inherits those computational advantages, but it is also prone to inherent problems such as z-fighting for incorrect depth bias.

Bibliography

[Annen et al. 08] Thomas Annen, Zhao Dong, Tom Mertens, Philippe Bakaert, Hans-Peter Seidel and Jan Kautz. "Real-Time All-Frequency Shadows in Dynamic Scenes." *ACM Transactions on Graphics (Proc SIGGRAPH)* 27:3 (2008), 34:1–34:8

[Bracewell 00] Ronald Bracewell. *The Fourier Transform and its Applications*. Singapore: McGraw-Hill International Editions, 2000.

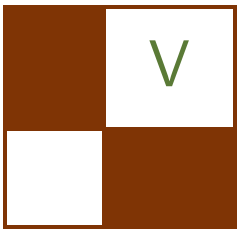
[Donnelly and Lauritzen 06] William Donnelly and Andrew Lauritzen. "Variance Shadow Maps." In *Symposium on Interactive 3D Graphics and Games*, pp. 161-165. New York: ACM, 2006.

[Fernando 05] Randima Fernando. "Percentage-Closer Soft Shadows." In *Proc SIGGRAPH Sketches*, pp. 35. New York: ACM, 2005.

[Gambau et al. 10] Jesus Gambau, Miguel Chover and Mateu Sbert. "Screen Space Soft Shadows." In *GPU Pro Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 477–491. Natick, MA: A K Peters, 2010.

[Reeves et al. 87] William Reeves, David Salesin and Robert Cook. "Rendering Antialised Shadows with Depth Maps." *Computer Graphics (Proc. SIGGRAPH)* 21:4 (1987), 283-291.

[Williams 78] Lance Williams. "Casting Curved Shadows on Curved Surfaces." *Computer Graphics (Proc. SIGGRAPH)* 12:3 (1978), 270–274.



V Handheld Devices

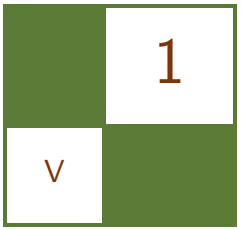
This part covers the latest development in programming GPUs of devices that are portable, such as mobile phones, personal organizers, and portable game consoles. The latest generation of GPUs for handheld devices comes with a feature set that is comparable to PC and console GPUs.

The first article, "A Shader-Based eBook Renderer," by Andrea Bizzotto illustrates a vertex-shader-based implementation of the page-peeling effect of a typical eBook renderer. It covers high-quality procedural antialiasing of the page edges, as well as some tricks that achieve a polished look. Two pages can be combined side-by-side to simulate a real book, and additional techniques are introduced to illustrate how to satisfy additional constraints and meet power-consumption requirements.

The second article of this part, "Post-Processing Effects on Mobile Devices," by Marco Weber and Peter Quayle describes a general approach to implement post-processing on handheld devices by showing how to implement a bloom effect with efficient convolution.

Joe Davis and Ken Catterall show in "Shader-Based Water Effects," how to render high-quality water effects at a low computational cost. Although there are many examples of water effects using shaders that are readily available, they are designed mainly for high-performance graphics chips on desktop platforms. This article shows how to tailor a technique discussed by Kurt Pelzer (in ShaderX^2, "Advanced Water Effects," 2004) to mobile platforms.

—Kristof Beets



A Shader-Based eBook Renderer

Andrea Bizzotto

Readers of eBooks are becoming increasingly popular. Touch screens and programmable GPUs, such as the POWERVR SGX Series from Imagination Technologies, can be combined to implement user-friendly navigation and page flipping functionality. This chapter illustrates a vertex-shader-based implementation of the page-peeling effect, and details some techniques that allow high-quality procedural antialiasing of the page edges, as well as some tricks that achieve a polished look. Two pages can be combined side-by-side to simulate a real book, and additional techniques are introduced to illustrate how to satisfy additional constraints and meet power consumption requirements.

1.1 Overview

The chapter is organized as follows: Section 1.2 introduces the mathematical model which is the basis of a page-peeling simulation, and shows how to use the vertex shader to render the effect with a tessellated grid on OpenGL ES 2.0 hardware. Section 1.3 discusses the additional constraints that need to be considered when rendering two pages side-by-side, and Section 1.4 illustrates some techniques that improve the visual look and deal with antialiasing. An approach that doesn't require a tessellated grid is illustrated in Section 1.5 to show how the technique can be adapted to work on OpenGL ES 1.1 hardware with minimal vertex overhead. Section 1.6 mentions some practical considerations regarding performance and power consumption. Finally, Sections 1.7, 1.8, and 1.9 discuss some aspects that have not been considered or that can be improved.

Throughout the article, points will be represented with a capital bold letter, vectors with a small bold letter, and scalars in italic. Page, quad, and plane will be used interchangeably to describe the same entity.

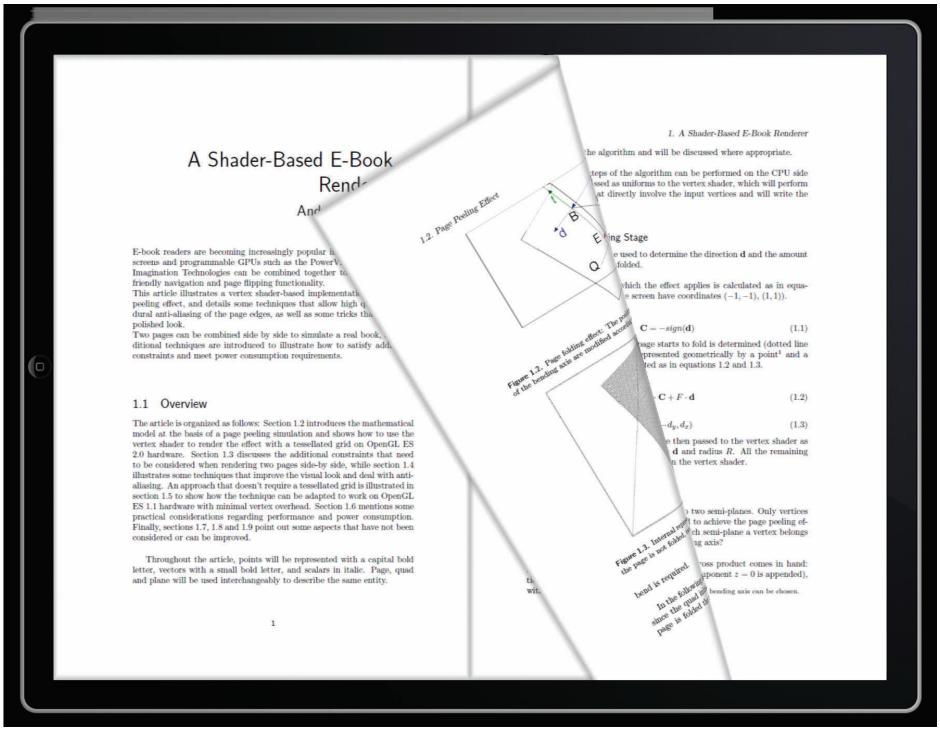


Figure 1.1. Page-folding effect rendered on screen.

1.2 Page-Peeling Effect

A page can be modeled as a quad. One corner can be selected and pulled in many directions. The mathematical model assumes that the interaction generates a bend on the page with a constant radius across the surface. When a page folds, it wraps around a semicircle by an angle that increases from 0 ° (no folding) to 180 ° (full semicircle). After that, the surface slides in parallel above the unfolded part of the page, as illustrated in Figures 1.1 and 1.2.

Since the page can be folded in any chosen direction, the original position of any arbitrary point on the quad can be modified.

In order to implement this effect on graphics hardware, the page can be represented as a highly tessellated grid whose vertices represent a discretized version of the points of the plane. The vertices' positions are modified in the vertex shader according to the folding algorithm, and the output values are then interpolated between vertices in the rasterization stage. As a smooth-looking bend is required, the tessellation factor needs to be high enough to simulate the nonlinear deformation with sufficient accuracy (as illustrated in Figure 1.3).

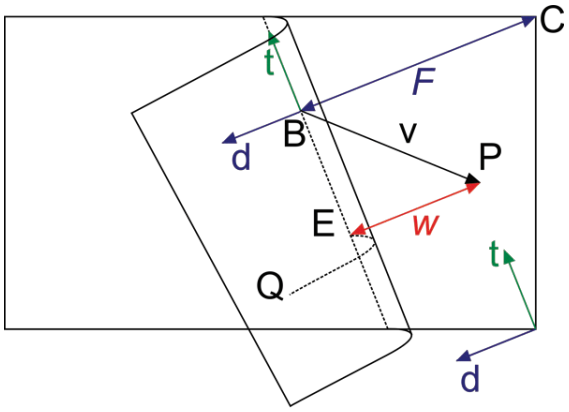


Figure 1.2. Page-folding effect: positions of the input vertices on the right of the bending axis are modified according to the model.

In the following discussion, all vectors can be considered two-dimensional since the quad initially lies on the plane with equation $z = 0$. When the page is folded, the z-coordinate does change, but this is not relevant for most stages of the algorithm and will be discussed where appropriate.

Some of the steps of the algorithm can be performed on the CPU side, and the results passed as uniforms to the vertex shader, which will perform only operations that directly involve the input vertices and will write the output position.

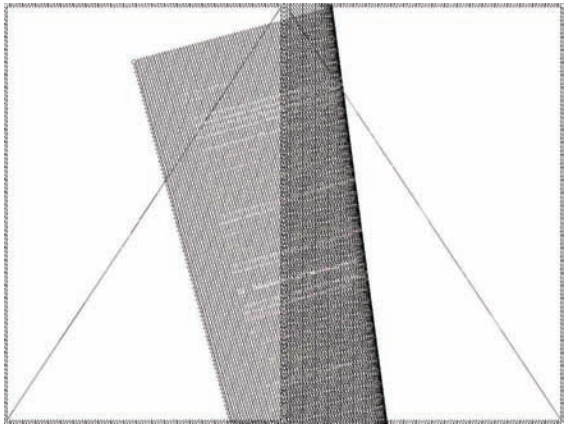


Figure 1.3. Internal representation: a tessellated grid is used as the input. When the page is not folded, two triangles can be used to render it.

1.2.1 Pre-Processing Stage

The touch screen can be used to determine the direction $\mathbf{d}$ and the amount (extent) F by which the page is folded.

First, the corner to which the effect applies is calculated as in Equation (1.1) (the corners of the screen have coordinates $(-1,-1)$, $(1,1)$):

$$\mathbf{C} = -\text{sign}(\mathbf{d}) \tag{1.1}$$

Then, the axis where the page starts to fold is determined (dotted line in Figure 1.2). This can be represented geometrically by a point[1] and a direction, which can be calculated as in Equations (1.2) and (1.3).

$$\mathbf{B} = \mathbf{C} + F \cdot \mathbf{d} \tag{1.2}$$

$$\mathbf{t} = (-d_y, d_x) \tag{1.3}$$

The calculated values $\mathbf{B}$ and $\mathbf{t}$ are then passed to the vertex shader as uniforms, together with the direction $\mathbf{d}$ and radius R. All the remaining steps of the algorithm are performed in the vertex shader.

1.2.2 Per-Vertex Stage

The bending axis partitions the plane into two semiplanes. Only vertices in the right semiplane need to be modified to achieve the page-peeling effect. How is it possible to determine to which semiplane a vertex belongs, and what is its distance from the bending axis?

The right-handedness property of the cross product comes in handy: given two vectors $\mathbf{t}$ and $\mathbf{v}$ (to which a third component $z = 0$ is appended), the resulting vector $\mathbf{s} = \mathbf{t} \times \mathbf{v}$ will point upward if $\mathbf{t}$ follows $\mathbf{v}$, and downward otherwise. Additionally, the length of $\mathbf{s}$ is given by Equation (1.4), where $\mathbf{t}$ is unitary:

$$|\mathbf{s}| = |\mathbf{t}||\mathbf{v}| \sin\ \theta_{tv} = |\mathbf{v}| \sin\ \theta_{tv} \tag{1.4}$$

Since $\mathbf{t}$ and $\mathbf{v}$ both lie on the same plane $z = 0$, the scalar value $w = s_z$, which satisfies $|w| = |s_z| = |\mathbf{s}|$, gives all the required information. In fact, the sign of w represents the direction of the resulting vector $\mathbf{s}$ and tells which semiplane the vertex belongs to, and its absolute value is the distance from the axis.

It can be noticed in Figure 1.2 that the relation $w = -\mathbf{d} \cdot \mathbf{v}$ also holds since the dot product calculates the projection of the vector $\mathbf{v}$ into $\mathbf{d}$, which can be on either side of the bending axis. To summarize, both the cross and dot product can be used to get the required information as Equation (1.5) shows:

$$|w| = |\mathbf{v}| \cos\ \theta_{dv} = |\mathbf{d} \cdot \mathbf{v}| = |\mathbf{t} \times \mathbf{v}| = |\mathbf{v}| \sin\ \theta_{tv} = |s_z|. \tag{1.5}$$

[1] For the purposes of the algorithm, any point on the bending axis can be chosen.

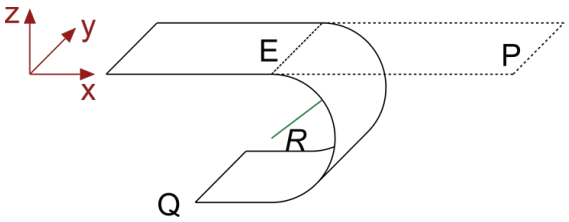


Figure 1.4. Model for bending effect: For vertices with $w > 0$ the position needs to to be recalculated by wrapping it along the curve path.

If a vertex lies on the right semiplane, it needs to be folded as illustrated in Figure 1.4.

The final vertex position $\mathbf{Q}$ needs to be updated according to the folding direction $\mathbf{d}$ and distance from the axis w. First, the projection $\mathbf{E}$ of $\mathbf{P}$ onto the bending axis is calculated as in Equation (1.6):

$$\mathbf{E} = \mathbf{B} + (\mathbf{v} \cdot \mathbf{t})\,\mathbf{t} \tag{1.6}$$

Figure 1.4 shows that the x-, y- and z-components of the input vertex are updated differently. All the remaining calculations are based on the known distance w and the direction $-\mathbf{d}$, labeled $\mathbf{u}$ for convenience. The angle $\alpha = w/R$ is used, as well, to update the final position.

If $\alpha < \pi$, then the final position of the vertex lies on the semicircle and can be calculated as in Equation (1.7):

$$\mathbf{Q_{xy}} = \mathbf{E} + R \cdot \sin\ \alpha \cdot \mathbf{u}. \tag{1.7}$$

If $\alpha \geq \pi$ instead, the updated position will just be parallel to the original plane as Equation (1.8) states:

$$\mathbf{Q_{xy}} = \mathbf{E} - (w - \pi R)\,\mathbf{u}. \tag{1.8}$$

As for the depth component, the final z-position is calculated as $Q_z = -2w/\pi$ if the vertex lies in the semicircle and $Q_z = -2R$ otherwise.

Note that the final depth value is always in the range allowed for normalized device coordinates (this is necessary to avoid clipping after the perspective divide), since the relation in Equation (1.9) holds:

$$Q_z \in (-2R, 0) \subset (-1, 1)\,, R < 0.5. \tag{1.9}$$

The described algorithm can be implemented in the vertex shader as shown in Listing 1.1. Note that the texture coordinates do not need to be altered since only the vertex positions change.

```
attribute highp    vec3  inVertex;
attribute mediump vec2  inTexCoord;
uniform highp float Radius;
uniform highp vec2 Direction;
uniform highp vec2 Tangent;
uniform highp vec2 Point;
varying mediump vec2   TexCoord;
const highp float PI = 3.141592;
const highp float INV_PI_2 = 2.0 / PI;
void main()
{
  highp vec2 vertex = inVertex.xy;
  highp vec2 v = vertex - Point;
  // w can equally be calculated with the cross product
  highp float w = -dot(v, Direction);
  if (w > 0.0)
  {
    highp vec2 E = Point + dot(v, Tangent) * Tangent;
    highp float angle = w / Radius;
    if (angle < PI) {
      gl_Position.xy = E - Radius * sin(angle) * Direction;
      gl_Position.z = -INV_PI_2 * w;
    }
    else {
      gl_Position.xy = E + (w - PI * Radius) * Direction;
      gl_Position.z = -2.0 * Radius;
    }
  }
  else
    gl_Position.xyz = vec3(vertex, 0.0);

  gl_Position.w = 1.0;

  TexCoord = inTexCoord;
}
```

Listing 1.1. OpenGL ES 2.0 vertex shader for basic page-peeling effect.

1.2.3 Taking Aspect Ratio into Account

The description above assumes that the input quad is a square. Since all devices have a rectangular aspect ratio, applying the basic algorithm will cause the image to appear stretched along the major axis and the edges of the page to look nonorthogonal. This can be solved by premultiplying the x-component of the input vertex by the aspect ratio before applying the algorithm, and dividing the x-component of the final position by the aspect ratio. Additionally, it is

recommended that the relative tessellation of the input grid matches the screen aspect ratio, so that the vertices will be spaced equally in the two dimensions, once stretched to the screen.

1.3 Enabling Two Pages Side-by-Side

The presented approach illustrates the basic page-folding effect, although a representative usage case is composed of two pages side-by-side, with the additional constraint that pages cannot be folded on the inner side. The algorithm can be extended to take this into account, and only the preprocessing stage needs to be modified.

More formally, the bending axis can intersect the page either on the top or bottom edge or both. The x-coordinate of this intersection (the closest to the inner edge if two intersections are present) represents the amount by which the page is folded on either the top or bottom edge, and must be smaller than the width of the page (which is equivalent to the aspect ratio a). Such amount x depends on the folding value F, direction $\mathbf{d}$, and tangent $\mathbf{t}$ as illustrated by Equation (1.10), which is solved by Equations (1.11) and (1.12):[2]

$$F\mathbf{d} + \lambda\mathbf{t} = \begin{pmatrix} x \\ 0 \end{pmatrix}, \tag{1.10}$$

$$\begin{cases} Fd_x - \lambda d_y = x \\ Fd_y + \lambda d_x = 0, \end{cases} \tag{1.11}$$

$$x = Fd_x - \left(-F\frac{d_y}{d_x}\right) d_y = F\left(d_x + \frac{d_y^2}{d_x}\right) = F\frac{d_x^2 + d_y^2}{d_x} = \frac{F}{d_x}. \tag{1.12}$$

If $x > a$, then F exceeds the maximum value given the direction $\mathbf{d}$, as shown in Figure 1.5(a). If a different angle is chosen, the page can fold in a direction that satisfies the constraint, while preserving the same folding value F.

Let θ be the angle corresponding to the direction $\mathbf{d}$. Values of $|\theta|$ close to $\pi/2$ cause x to approach the limit a quickly, since the page folds almost vertically,[3] therefore when $x > a$ it is appropriate to use a new value θ', where $|\theta'| < |\theta|$. Once the new angle is calculated, all other dependent variables can be updated and passed to the vertex shader.

[2] Here the corner is assumed to have coordinates $(0, 0)$ and applying the two vectors $F\mathbf{d}$ and $\lambda\mathbf{t}$ corresponds to a translation of $(x, 0)^T$.

[3] Since the page can fold upwards or downwards, θ can be positive or negative and the modulus operator accounts for this.

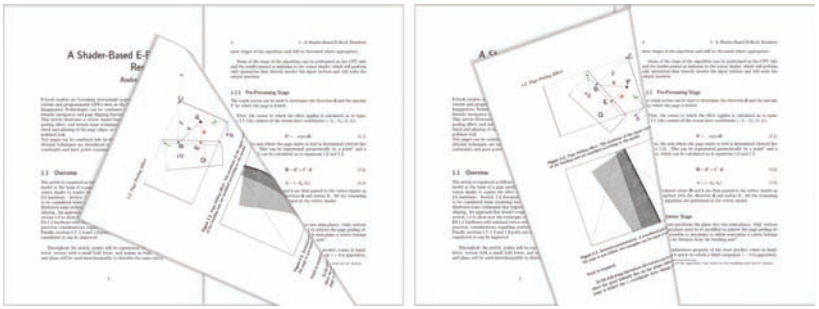


Figure 1.5. Angle correction: to prevent page tearing (left), the direction is modified in order to meet the constraint $x \leq a$ (right).

The correction $\theta_c = \theta' - \theta$ that needs to be applied in order to meet the constraint is calculated in Equation (1.13):

$$\theta_c = -\text{sign}(d_x d_y) F \frac{x - a}{x}. \tag{1.13}$$

The term $-\text{sign}(d_x d_y)$ takes into account the fact that θ can be positive or negative, ensuring that $|\theta'| < |\theta|$.[4]

The absolute value of θ_c needs to be proportional to F and the difference $(x - a)$, normalized by a factor $1/x$ to ensure small corrections for large x values (this is an heuristic that works well in practice).

```
bool PageFolded(theta, newTheta, radius, newRadius) {
  if (theta * newTheta < 0) {
    newRadius = radius - abs(newTheta);
    newTheta = 0;
    if (newRadius < 0) {
      newRadius = 0;
      return true; // page completely folded
    }
  }
  return false; // page not completely folded
}
```

Listing 1.2. Pseudo-code to enforce the additional constraint on side-by-side pages.

[4]The term d_x in Equation (1.13) is necessary to handle correctly the right page, where the angle correction needs to be inverted. The full example can be found in the code.

If F is large enough, it means that the user is dragging a finger across the whole screen, and the page should fold completely. In order to do this it is necessary to modify the current model so that once θ' reaches 0 (condition by which the page is parallel and almost completely covers the one underneath), the radius decreases to 0 as well to complete the peel effect. Listing 1.2 illustrates the final update stage.

Once the angle and radius are updated, all the required uniforms are calculated and passed to the vertex shader.

1.4 Improving the Look and Antialiasing Edges

Since pages are predominantly white, when a page is folded the edge is not clearly visible, and it's therefore useful to add some sort of shading to improve the general look. A simple technique to achieve this is to use a fade-to-gray gradient on the edges of the pages. This can be done efficiently by splitting the grid representing a page into two meshes (as in Figure 1.3), one representing the page content to be rendered with a standard lookup shader, and a border rendered with a shader that mixes the color of the page with a gray gradient (since a color mix is done in the fragment shader and this affects only the border of the page, the simpler shader can be used when the gradient is not necessary).

The effect highlights the border of the pages as required, but introduces some noticeable aliasing artifacts on the edges. By adding an external border that can be rendered with a similar gradient and enabling blending, a shadow-like effect is made and the aliasing problem is mitigated. Figure 1.6 shows how the borders can improve the render quality.

Even though aliasing is sensibly reduced by the introduction of the external border, it is not completely removed in cases where the destination color is not white. This is noticeable on the black title text in Figure 1.6(c), since, on the edge, the internal and external fragments have intensity 0.5 and 0.0, respectively, the latter due to the blend between the black text and the gray border. The quality can be considered acceptable and further refinements are not considered here.

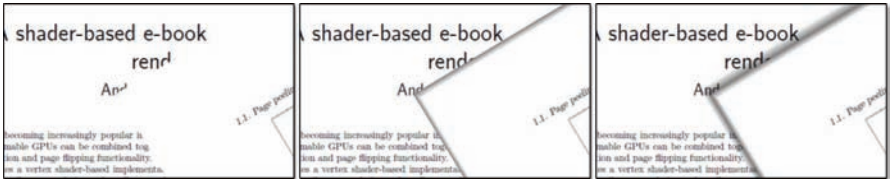


Figure 1.6. Antialiasing edges: simple shader (left), internal gradient border (center), internal and external border (right). The intensity values and widths of the borders can be tweaked to obtain the desired look.

In addition to the physical borders of the page, the area where the page bends includes an edge that can be perceived visually and could be improved by some form of shading. A gradient-based technique can be used here too, and the gradient can be calculated as a simple function of the angle α, available in the vertex shader. Since the folding direction is arbitrary, in this case it's not possible to split the original mesh in two, and this results in a more expensive fragment shader being applied to the whole page. The next section introduces an alternative rendering technique that, as a side effect, addresses this issue.

1.5 Direction-Aligned Triangle Strip

By using a screen-aligned grid, the tessellation needs to be uniform across the whole page, even though only a small part of it is folded. While in principle this can enable more complex deformation effects in the vertex shader, it is not strictly necessary for a simple page fold. A more efficient approach that exploits the nature of the problem is to tessellate only the bent part of the page, while two trapezoids can be used to render the flat areas. Since the fold is one-dimensional (i.e., it doesn't vary along the tangent) it is sufficient to generate a triangle strip aligned with the direction $\mathbf{d}$.

As Figure 1.7 shows, in the general case the page can be split into two trapezoids and a triangle strip. The intersection between the first trapezoid and the strip is delimited by the bending axis, while the length of the strip equals the length of the semicircle πR. The method in [Sutherland and Hodgman 74] can be used as a general algorithm to clip the page against the arbitrary bending

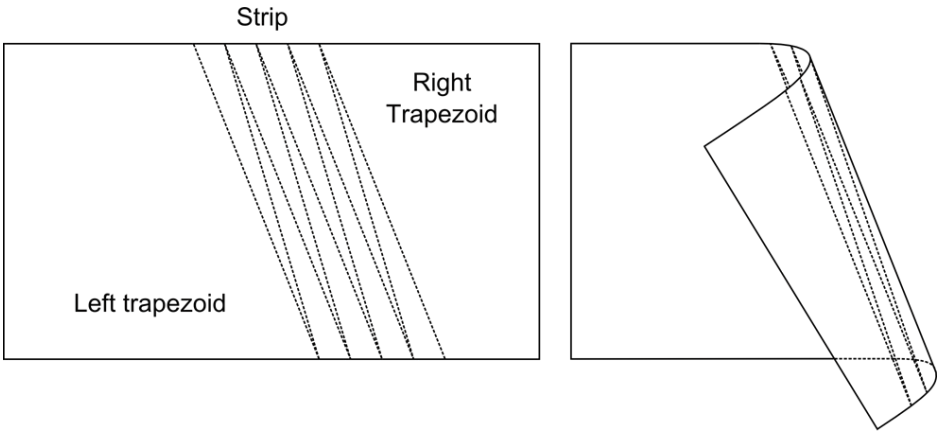


Figure 1.7. Direction-aligned strip: by using tessellation on only the curled part of the page, the overall number of vertices can be sensibly reduced. Additionally, the grid can be simplified to a triangle strip.

axis and create the two trapezoids. The triangle strip and the trapezoids can be generated procedurally on the CPU every time the input changes and can then be passed to OpenGL for rendering.

If an internal border is used as described in the previous section, the number of meshes and intersections to be determined further increases. Many subcases can be envisioned depending on how the bending axis partitions the page, and some extensive coding is necessary to cover all of them. If needed, the whole geometry can be preprocessed on the CPU and the page can be submitted already folded to the graphics renderer, allowing this technique to run on OpenGL ES 1.1-class hardware. For the purposes of this article only the general idea is presented and the implementation is left to the reader (though code for the tessellation-grid approach is included).

Note, separating the input quad into individual meshes further reduces the overall number of computations performed on the GPU, since a simple lookup shader can be used for most of the area (flat trapezoids), while the application of fancier shaders with gradients can be limited to the bent part of the page.

1.6 Performance Optimizations and Power Consumption

To ensure maximum battery life, render-on-demand is implemented by updating the frame only when the folding value and direction change. Although very simple to implement, this technique is critical for this kind of application, since expected battery life for eBook readers is very important. Rendering the same static image in each frame needlessly drains the battery of the device.

In typical usage the user reads a page, then folds, then reads another page and so on. Most of the time will be spent idle waiting for the next page to be folded. Drawing nonfolded pages can be done very cheaply by rendering simple quads.

When the page is actually folding and a render is required, it is preferable to use the triangle strip variant instead of the tessellated grid approach, since many fewer vertices are submitted. The cost difference between the two techniques does not apply if a page is partially folded but no render is required.

1.7 Putting it Together

The techniques illustrated so far can be used to render two pages side-by-side and fold them if required. By simply adding two more pages and keeping track of the textures to be bound to each page, it is possible to simulate a book with an arbitrary number of pages. Additional coding is necessary to limit the regions of the screen where the user can interact (pages can be folded only from the borders) and to handle page unfolding correctly. The two sides of a page can be

represented by two different textures, and the page can be rendered twice, with front- and back-face cull. Figure 1.1 shows the final effect rendered on screen.

The available code features a simple eBook application that allows the user to browse through a predefined set of pages and takes into account some additional practical conditions not considered in this article.

1.8 Future Work

The article has been presented with the assumption that the pages to be rendered are stored as textures, and no considerations have been made about the best way to render text. True-type fonts are common in eBook readers and a better way to display text could be to render true-type text to textures whose resolutions match the area on screen that is covered by the pages. More research could be done to ensure the best mapping of the text on screen even where the page is folded.

Procedural antialiasing techniques proved to be effective, and tweaking the blending values and gradients resulted in a neat look. Some additional improvements can be achieved, for example, by procedurally mixing the gradient in the external border with the contents underneath it, rather than using simple alpha blending.

Touchscreen interaction has been used mainly to determine the direction in which the page needs to be folded. A more advanced use could be to enable zooming and additional features based on multitouch input.

The mathematical model used to enable the page-peeling effect is simple, and some constraints like parallelism between the unfolded and folded parts of the page, and constant folding radius across the page could be relaxed. More complex models could investigate how to use multitouch to apply more folds to a single page or enable more complex types of deformation, where the tessellated-grid approach would be more suitable than the tailored triangle-strip variant.

1.9 Conclusion

Graphics hardware capable of vertex processing can exploit the problem of folding a plane in any arbitrary direction by means of a highly tessellated grid. A sample application for iPad (available in the source code release) has been developed, and approximately 17,000 faces have been used per-page without any noticeable degrade in visual quality or performance at 1024×768 resolution.

The cost per vertex is relatively low, given the optimizations introduced in this paper, and the fragment shaders are quite simple, generally consisting of a texture lookup and a sum or mix with a gradient color, making low-end OpenGL ES 2.0 devices a good target for this kind of application.

By taking a different approach, it is possible to reduce the input of the graphics renderer to two trapezoids and a triangle strip, which can be rendered on OpenGL ES 1.1-class hardware, although more coding is required to handle all the intersection subcases and pregenerate the geometry.

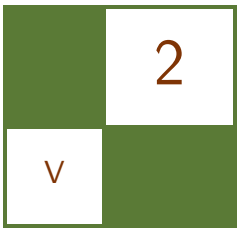
Render-on-demand is central in this design to maximize battery life, and some expedients can be used to minimize the cost-per-render.

1.10 Acknowledgments

The author would like to thank Kristof Beets, Sergio Tota, and the whole BizDev and DevTech teams for their work, contribution and feedback.

Bibliography

[Sutherland and Hodgman 74] Ivan E. Sutherland and Gary W. Hodgman. "Reentrant Polygon Clipping." *Communications of the ACM* 17:1 (1974), 32–42.



Post-Processing Effects on Mobile Devices

Marco Weber and Peter Quayle

Image processing on the GPU, particularly real-time post-processing, is an important aspect of modern graphical applications, and has wide and varied uses from games to user interfaces. Because of the continually improved performance levels of graphics hardware, post-processing effects are now common on the desktop and in the console space. With the current generation of OpenGL ES 2.0-enabled embedded GPUs, an increasing number of such effects are also possible on mobile devices. Bloom effects, for example, can add a vivid glow to a scene; color transforms and distortions can provide a stylistic flair; blur, and depth-of-field effects can draw attention to areas of interest, or highlight selected elements in a user interface. Figure 2.1 shows some generic examples of post-processing effects, and Figure 2.2 demonstrates the use of a blur effect in a user interface.

Post-processing is the modification and manipulation of captured or generated image data during the last stage of a graphics pipeline, resulting in the final output picture. Traditionally, these operations were performed by the host system CPU by reading back the input image data and altering it. This is a very costly operation and is not suitable for applications that require interactive frame rates on mobile platforms. Using the processing power of modern graphics hardware to do the image data manipulation is a more practical and efficient approach.

This chapter describes a general approach to post-processing and provides an example of a real-time effect that can be implemented on OpenGL ES 2.0-capable hardware, such as POWERVR SGX.

2.1 Overview

The general idea behind post-processing is to take an image as input and generate an image as output (see Figure 2.3). You are not limited to only using the provided input image data, since you can use any available data source (such as

Figure 2.1. Radial twist, sepia color transformation, depth of field, depth of field and sepia color transformation combined (clockwise from top left).

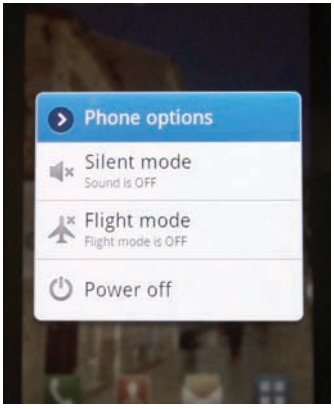


Figure 2.2. Post-processing effect on the Samsung Galaxy S. The background of the user interface is blurred and darkened to draw attention to items in the foreground.

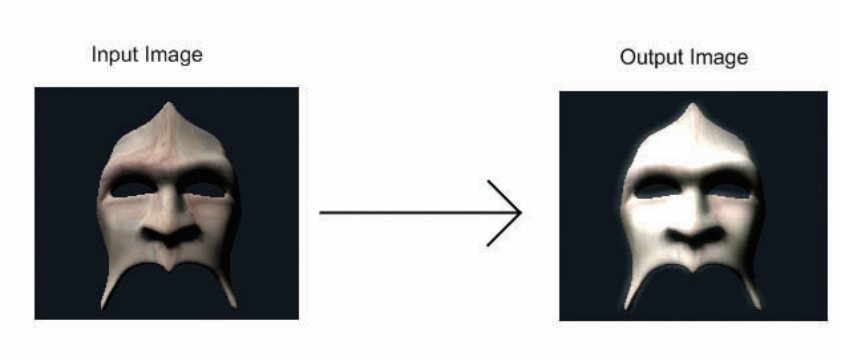


Figure 2.3. Image input and output.

depth and stencil buffers), as additional input for the post-processing step. The only requirement is that the end result has to be an image.

One direct advantage of this approach is that, due to the identical basic format of input and output, it is possible to chain post-processing techniques. As illustrated in Figure 2.4, the output of the first post-processing step is reused as input for the second step. This can be repeated with as many post-processing

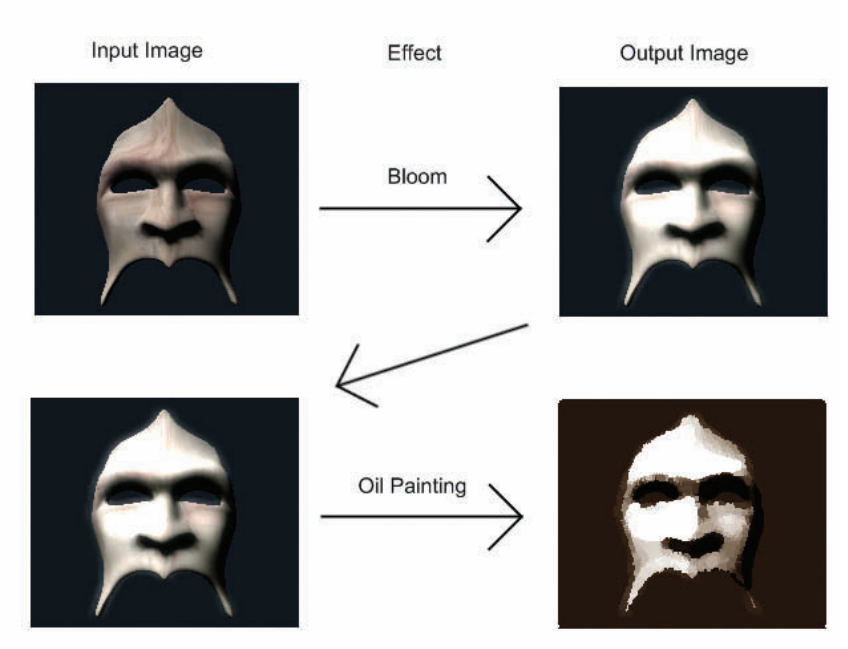


Figure 2.4. Image input and output chain.

techniques as required. Some sets of post-processing techniques can be merged to avoid the overhead of chaining, while others can exploit chaining to achieve a performance increase, as explained later.

Each step is performed by rendering to a frame buffer object (FBO). The final output image will be displayed on the screen.

2.2 Technical Details

To generate the first input image we must render our scene to a texture. As mentioned in the previous section, to do this we use FBOs. There are other ways to accomplish the same results, such as creating textures by copying data from the framebuffer, but using FBOs is better for performance (for a more detailed discussion see [Imagination Technologies 10]).

We will make use of pixel-shader support, as found in the POWERVR SGX graphics cores, since they provide a great deal of flexibility when developing post-processing effects. It is possible to implement some basic post-processing techniques with a fixed-function pipeline, but this is beyond the scope of this article.

The basic algorithmic steps for a post-processing effect are as follows:

1. Render the scene to a texture.
2. Render a full screen pass using a custom pixel shader, with the texture from the previous stage as input, to apply the effect.
3. Repeat step two until all effects are processed.

The first step is the most straightforward, because it simply requires setting a different render target. Instead of rendering to the back buffer, you can create an FBO of the same dimensions as your frame buffer. In the case that the frame buffer dimensions are not a power of two (e.g., 128×128, 256×256 etc.), you must check that the graphics hardware supports non-power-of-two (NPOT) textures. If there is no support for NPOT textures, you could allocate a power-of-two FBO that approximates the dimensions of the frame buffer. For some effects it may be possible to use an FBO far smaller than the frame buffer, as discussed in Section 2.4.

In step two, the texture acquired during step one can be used as input for the post-processing. In order to apply the effect, a full-screen quad is drawn using a post-processing pixel shader to apply the effect to each pixel of the final image.

All of the post-processing is executed within the pixel shader. For example, in order to apply an image convolution filter, neighboring texels have to be sampled and modulated to calculate the resulting pixel. Figure 2.5 illustrates the kernel, which can be seen as a window sliding over each line of the image and evaluating each pixel at its center by fetching neighboring pixels and combining them.

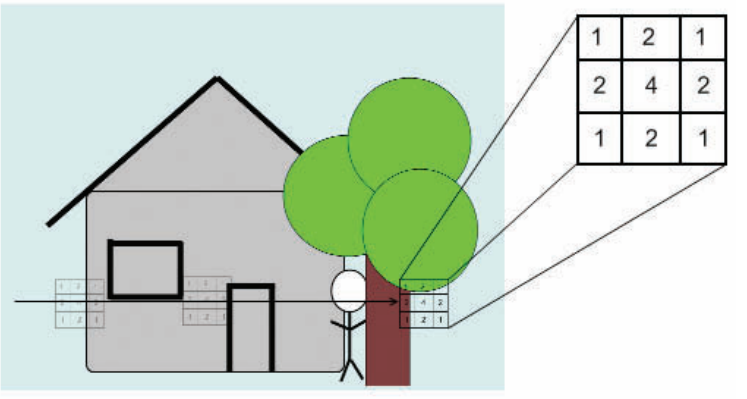


Figure 2.5. Gaussian blur filtering.

Step three describes how easy it is to build a chain of post-processing effects by simply executing one after another, using the output of the previous effect as the input for the next effect. In order to accomplish this, it is necessary to allocate more than one frame buffer object as it is not possible to simultaneously read from and write to the same texture.

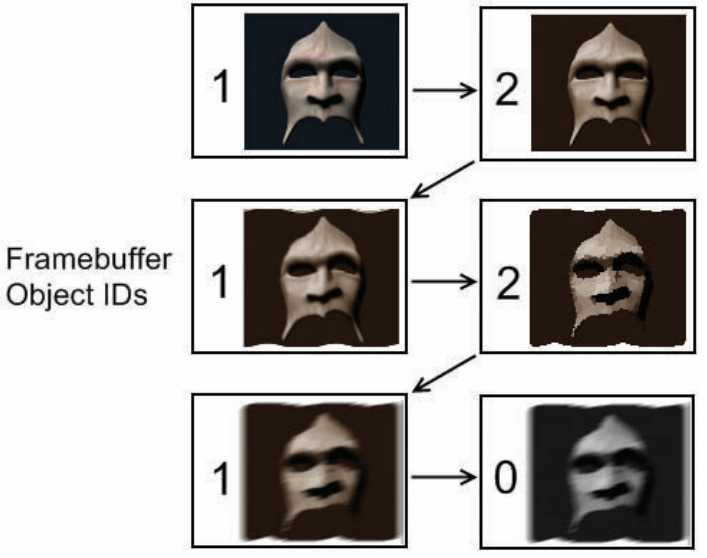


Figure 2.6. Post-processing steps and frame buffer IDs.

Figure 2.6 illustrates the re-use of several frame buffer objects:

- The initial step renders to the frame buffer object with ID 1
- The second step renders to the frame buffer object with ID 2, using the previous frame buffer object as input.
- The whole procedure is repeated for steps three and four, but instead of using frame buffer object 2 again for the last step, the back buffer is used since the final result will be displayed on the screen.

Depending on the individual effect, some of the illustrated passes may be merged into one pass. This avoids the bandwidth cost associated with processing the whole image for each effect that is merged, and reduces the total number of passes required.

2.3 Case Study: Bloom

The whole concept of post-processing, as presented in the previous section, is suitable for high-performance graphics chips in the desktop domain. In order to implement post-processing on mobile graphics chipsets, such as POWERVR SGX graphics cores, it is most important to act with caution. In this section, we illustrate an actual implementation of the bloom effect tailored for implementation on an embedded GPU, such as POWERVR SGX, at an interactive frame rate. The required optimizations and alterations to the original algorithm are explained throughout the following sections. At the beginning of this section the effect itself will be explained, followed by the actual implementation and some optimization strategies.

Figure 2.7. Mask without and with glow applied.

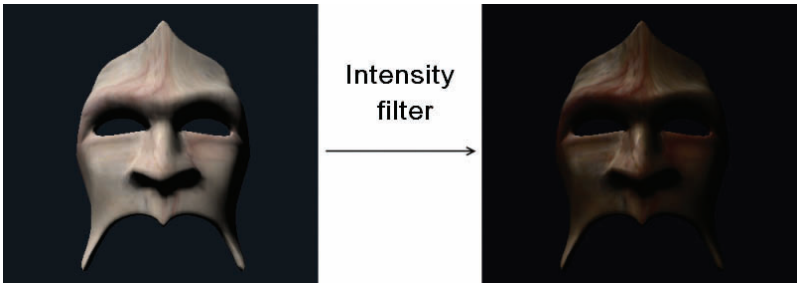


Figure 2.8. Applying an intensity filter to obtain bright parts.

The bloom effect simulates our perception of bright light by producing a pronounced glow and a halo around bright objects. Furthermore, it gives the impression of high dynamic range rendering despite being done in a low dynamic range. Another useful side effect is that it reduces aliasing artifacts due to the slight glow at the edges of objects. The whole effect and its intensity can be controlled by an artist and can be used to attract attention to objects and make them distinct (see Figure 2.7).

The bloom effect is achieved by intensifying bright parts of the image. In order to accomplish this, the bright parts of the image have to be identified and isolated. This can happen either implicitly by applying an intensity filter to the input image to extract the bright parts (see Figure 2.8), or explicitly by specifying the glowing parts through a separate data source, (e.g., the alpha channel of the individual textures).

The intensity filtered texture, which contains the bright parts of the image, will then be blurred with a convolution kernel in the next step. The weights of the kernel used in this example are chosen to resemble the weights of the Gaussian blur kernel. The kernel itself is applied by running a full shader pass over the whole texture and executing the filter for each texture element. Depending on the number of blur iterations and the size of the kernel, most of the remaining high frequencies will be eliminated and a ghostly image will remain (see Figure 2.9). Furthermore, due to the blurring, the image is consecutively smeared and thus enlarged, creating the halos when combined with the original image.

The final step is to additively blend the resulting bloom texture over the original image by doing a full screen pass. This amplifies the bright parts of the image and produces a halo around glowing objects due to the blurring of the intensity-filtered texture (see Figure 2.10).

The final amount of bloom can be controlled by changing the blend function from additive blending to an alpha-based blending function, offering even more artistic freedom. By animating the alpha value we can vary the amount of bloom and simulate a pulsing light.

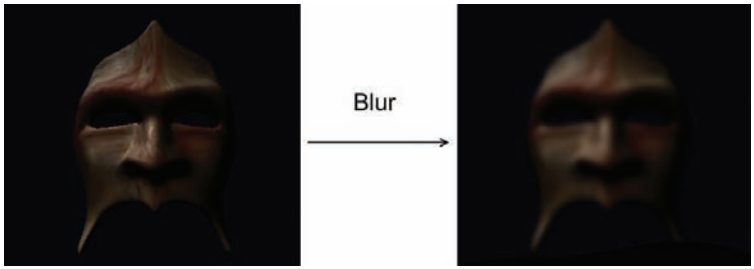


Figure 2.9. Applying Gaussian blur.

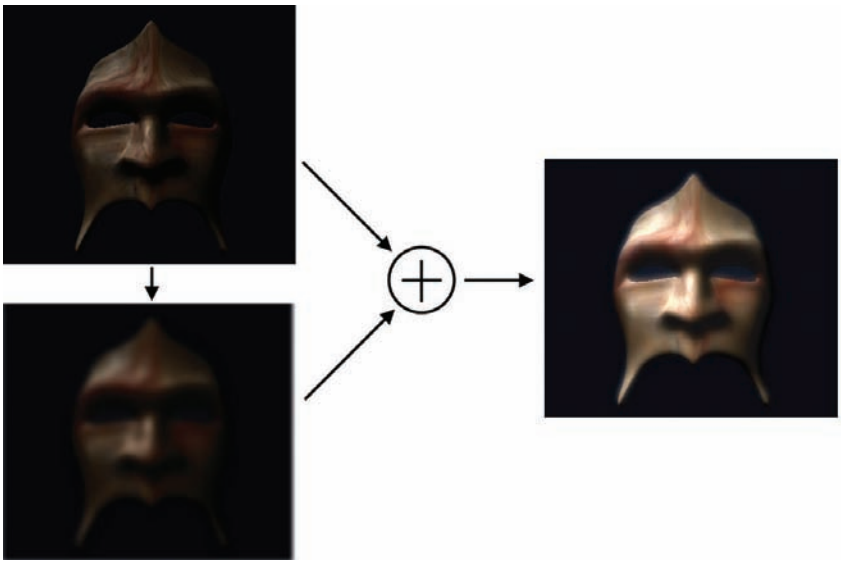


Figure 2.10. Additive blending of original and blurred intensity-filtered image.

2.4 Implementation

The bloom algorithm presented in the previous section describes the general approach one might implement when processing resources are vast. Two full screen passes for intensity filtering and final blending and several passes for the blur filter in the most naive implementation are very demanding even for the fastest graphics cards.

Due to the nature of mobile platforms, adjustments to the original algorithm have to be made in order to get it running, even when the hardware is equipped with a highly efficient POWERVR SGX core.

The end result has to look convincing and must run at interactive frame rates. Thus, most of the steps illustrated in Section 2.3 have to be modified in order to meet the resource constraints.

2.4.1 Resolution

One of the realities of mobile graphics hardware is a need for low power and long battery life, which demand lower clock frequencies. Although the POWERVR SGX cores implement a very efficient tile-based deferred rendering approach, it is still essential to optimize aggressively when implementing full screen post-processing effects.

In our implementation the resolution of the frame buffer object for the blurred texture was set to 128×128, which has shown to be sufficient for VGA (640×480) displays. Depending on the target device's screen and the content being rendered, even 64×64 may be adequate; the trade-off between visual quality and performance should be inspected by regularly testing the target device. It should be kept in mind that using half the resolution (e.g., 64×64 instead of 128×128) means a 75% reduction in the number of pixels being processed.

Since the original image data is not being reused because of the reduced resolution, the objects using the bloom effect have to be redrawn. This circumstance can be exploited for another optimization. As we are drawing only the objects which are affected by the bloom, it is possible to calculate a bounding box enclosing these objects that in turn will be reused in the following processing steps as a kind of scissor mechanism.

When initially drawing the objects to the frame buffer object, one could take the optimization even further and completely omit texture mapping (see Figure 2.11). This would mean that the vertex shader would calculate only the

Figure 2.11. Difference between nontextured (left) and textured (right) bloom.

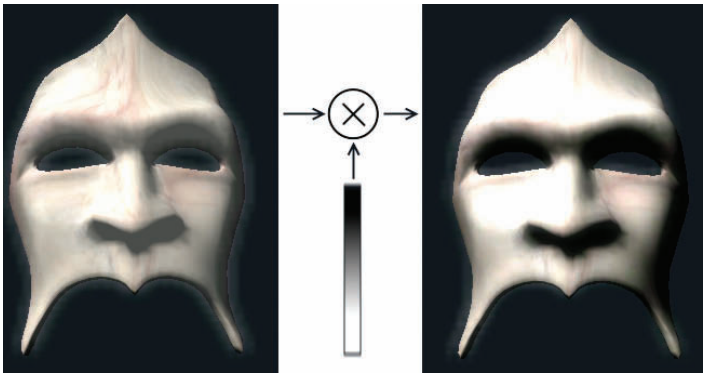


Figure 2.12. Transforming the brightness scalar by doing a texture lookup in an intensity map.

vertex transformation and the lighting equation, which would reduce the amount of data being processed in the fragment shader even further, at the expense of some detail. Each different case should be evaluated to judge whether the performance gain is worth the visual sacrifice.

When omitting texture mapping, the scalar output of the lighting equation represents the input data for the blur stage, but if we simply used scalar output for the following steps, the resulting image would be too bright, even in the darker regions of the input image, which is why the intensity filter has to be applied. Applying the intensity filter can be achieved by doing a texture lookup into a 1D texture, representing a transformation of the luminance (see Figure 2.12). This texture can be generated procedurally by specifying a mapping function, or manually, whereby the amount of bloom can be stylized to meet artistic needs. The lookup to achieve the intensity filtering can potentially be merged into the lighting equation. Other parts of the lighting equation could also be computed via the lookup table, for example, by premultiplying the values in it by a constant color.

2.4.2 Convolution

Once we've rendered our intensity-filtered objects to the frame buffer object, the resulting image can be used as input for the blur-filtering steps. This section explains the blur-filtering methods which are depicted in Figure 2.13.

Image convolution is a common operation and can be executed very efficiently on the GPU. The naive approach is to calculate the texture-coordinate offsets (e.g., 1/width and 1/height of texture image) and sample the surrounding texels. The next step is to combine these samples by applying either linear filters (Gaussian, median, etc.) or morphologic operations (dilation, erosion, etc.).

Figure 2.13. Blurring the input image in a low-resolution render target with a separated blur filter kernel.

In this case we will apply a Gaussian blur to smooth the image. Depending on the size of the filter kernel, we have to read a certain amount of texture values, multiply each of them by a weight, sum the results, and divide by a normalization factor. In the case of a 3×3 kernel this results in nine texture lookups, nine multiplications, eight additions, and one divide operation, which is a total of 27 operations to filter a single texture element. The normalization can be included in the weights, reducing the total operation count to 26.

Fortunately, the Gaussian blur is a separable filter, which means that the filter kernel can be expressed as the outer product of two vectors:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = (1 \quad 2 \quad 1) \otimes (1 \quad 2 \quad 1).$$

Making use of the associativity,

$$t \cdot (v \cdot h) = (t \cdot v) \cdot h,$$

where t represents the texel, v the column, and h the row vector, we can first apply the vertical filter and, in a second pass, the horizontal filter, or vice versa. This results in three texture lookups, three multiplications, and two additions per pass, giving a total of 16 operations when applying both passes. This reduction in the number of operations is even more dramatic when increasing the kernel size (e.g., 5×5, 7×7, etc.)(see Table 2.1):

Kernel	Texture Lookups	Muls	Adds	No. Of Operations
3x3 (standard)	9	9	8	26
3x3 (separated)	6	6	4	16
5x5 (standard)	25	25	24	74
5x5 (separated)	10	10	8	28
9x9 (standard)	81	81	80	242
9x9 (separated)	18	18	17	53

Table 2.1.

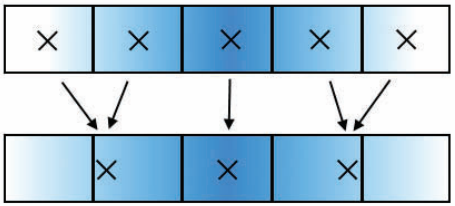


Figure 2.14. Reducing the number of texture lookups by using hardware texture filtering.

In most cases, separating the filter into horizontal and vertical passes results in a large performance increase. However, the naive single-pass version may be faster in situations in which bandwidth is severely limited. It is always worth benchmarking to ensure the best solution for a given platform or scenario.

The number of texture lookups can be decreased again by exploiting hardware texture filtering. The trick is to replace the texture lookup for the outer weights with one which is between the outer texels, as shown in Figure 2.14.

The way this works is as follows: when summing the contribution, s, of the outer texels, t_0 and t_1, in the unoptimized version, we use

$$s = t_0 w_0 + t_0 w_1. \tag{2.1}$$

When we sample between the outer texels with linear filtering enabled we have

$$s = t_0(1-u) + t_1 u, \tag{2.2}$$

where u is the normalized position of the sample point in relation to the two texels. So by adjusting u we can blend between the two texel values. We want to blend the texels with a value for u such that the ratio of $(1-u)$ to u is the same as the ratio of w_0 to w_1. We can calculate u using the texel weights

$$u = w_1 / (w_0 + w_1). \tag{2.3}$$

We can then substitute u into Equation (2.2). Because u must be a value between 0 and 1, we need to multiply s by the sum of the two weights. Our final equation looks like this:

$$s = (t_0 (1-u) + t_1 u)(w_0 + w_1). \tag{2.4}$$

Although this appears to contain more operations than Equation (2.1), the cost of the term in the first set of brackets is negligible because linear texture filtering is effectively a free operation. In the case of the 5×5 filter kernel, the number of texture lookups can be reduced from ten to six, yielding the identical number of computation necessary as for the 3×3 kernel.

It is important that the texture coordinates are calculated in the vertex shader and passed to the pixel shader as varyings, rather than being calculated in the pixel shader. This will prevent dependent texture reads. Although these are supported, they incur a potentially substantial performance hit. Avoiding dependent texture reads means that the texture-sampling hardware can fetch the texels sooner and hide the latency of accessing memory.

2.4.3 Blending

The last step is to blend the blurred image over the original image to produce the final result, as shown in Figure 2.15.

Therefore, the blending modes have to be configured and blending enabled so that the blurred image is copied on top of the original one. Alternatively, you could set up an alpha-value-based modulation scheme to control the amount of bloom in the final image.

In order to increase performance and minimize power consumption, which is crucial in mobile platforms, it is best that redundant drawing be avoided as much as possible. The single most important optimization in this stage is to minimize the blended area as far as possible. Blending is a fill-rate intensive operation, especially when being done over the whole screen. When the bloom effect is applied only to a subset of the visible objects, it is possible to optimize the final blending stage:

- In the initialization stage, calculate a bounding volume for the objects which are affected.
- During runtime, transform the bounding volume into clip space and calculate a 2D bounding volume, which encompasses the projected bounding volume. Add a small margin to the bounding box for the glow.

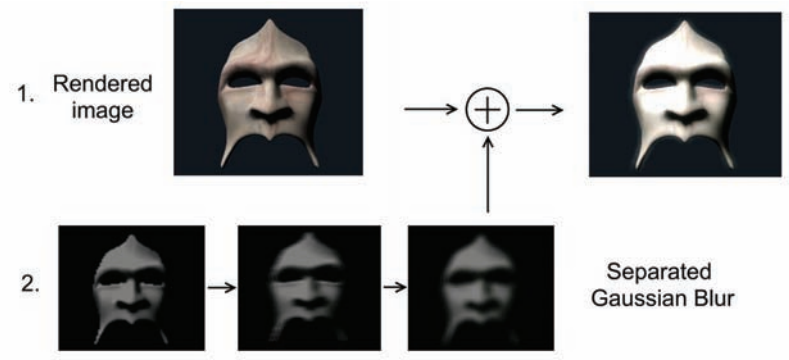


Figure 2.15. Overview of the separate bloom steps.

Figure 2.16. Bounding box derived rectangle (red) used for final blending.

- Draw the 2D bounding object with appropriate texture coordinates to blend the blurred texture over the original image.

Figure 2.16 shows the blending rectangle which is derived from the object's bounding box. The bounding box in this case is a simple axis-aligned bounding box which is calculated during the initialization. At runtime, the eight vertices of the bounding box are transformed into clip space and the minimum and maximum coordinate components are determined. The blended rectangle is then derived from these coordinates and the texture coordinates are adapted to the vertex positions. Depending on the shape of the object, other, more suitable, bounding volumes might be appropriate (e.g., bounding spheres).

This bounding-box-directed blending can lead to artifacts when the blending rectangles of two or more objects overlap, resulting in sharp edges and highlights that are too bright. A work-around for this overlap issue is to use the stencil buffer:

- Clear the stencil buffer to zero and enable stencil testing.
- Configure the stencil test so that only the pixels with a stencil value of zero are rendered, and the stencil value is always incremented.
- Draw all bounding volumes and disable stencil test.

Use of this algorithm prevents multiple blend operations on a single pixel and produces the same result as a single full screen blend. On a tile-based deferred renderer like POWERVR SGX, stencil operations are low cost.

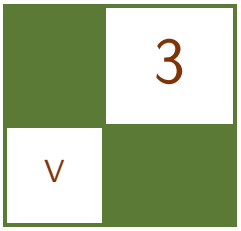
2.5 Conclusion

We have presented a brief introduction to post-processing, followed by a detailed case study of a well-known post-processing effect. We have illustrated optimization techniques that make it possible to use the effect while achieving interactive

framerates on mobile hardware. Many of the optimization techniques shown can be adapted and applied to other post-processing effects.

Bibliography

[Imagination Technologies 10] *PowerVR SGX OpenGL ES 2.0 Application Developer Recommendations*. 2010. http://www.imgtec.com/POWERVR/insider/POWERVR-sdk.asp.



Shader-Based Water Effects

Joe Davis and Ken Catterall

3.1 Introduction

Generating efficient and detailed water effects can add a great deal of realism to 3D graphics applications. In this chapter, we highlight techniques that can be used in software running on POWERVR SGX-enabled platforms to render high-quality water effects at a relatively low computational cost.

Such effects can be achieved in a variety of ways, but we will focus on the use of vertex and fragment shaders in OpenGL ES 2.0 to alter the appearance of a plane to simulate a water effect.

Although there are many examples of water effects using shaders that are readily available, they are designed mainly for high-performance graphics chips on desktop platforms. The following sections of this chapter describe how the general concepts presented in desktop implementations, in particular the technique discussed by K. Pelzer [Pelzer 04], can be tailored to run at an interactive frame rate on even low-cost POWERVR SGX platforms, including the optimizations that were made to achieve the required performance.

We refer to an example application OGLES2Water that is part of the freely available POWERVR OpenGL ES 2.0 SDK, which is included in the example code with this article. Up-to-date SDKs are available from the Imagination Technologies website.[1] Specific performance numbers cited refer to tests run on an OMAP3530 BeagleBoard[2] platform at VGA resolution.

3.2 Techniques

3.2.1 Geometry

In the demonstration, the plane that the water effect is applied to is a horizontal plane in world space, extending to the boundaries of the view frustum—this is

[1] http://www.imgtec.com/powervr/insider/powervr-sdk.asp
[2] http://www.beagleboard.org

constructed from only four or five points, as a high level of tessellation is not required. The `PVRTools` library from the POWERVR SDK contains a function, `PVRTMiscCalculateInfinitePlane()`, which obtains these points for a given set of view parameters. Because the plane is horizontal, certain calculations can be simplified by assuming the normal of the plane will always lie along the positive y-axis.

A skybox is used to encapsulate the scene. This is textured with a PVRTC-compressed 4 bits per-pixel format cubemap using bilinear filtering with nearest mipmapping to provide a good balance between performance and quality (for more information, see S. Fenney's white paper on texture compression [Feneny 03]).

3.2.2 Bump Mapping

To simulate the perturbation of the water's surface, a normal map is used to bump the plane. The normal map used in the demo is y-axis major (as opposed to many other bump maps that are z-axis major). The texture coordinate for each vertex is calculated in the vertex shader as the x- and z-values of the position attribute that has been passed into the shader. The water surface is animated by passing a time-dependent offset to the bump-map coordinates.

Since this offset amounts to a simple linear translation of the bump map, the effect on its own looks unrealistic because the perturbation travels in a single direction, rather than rippling as one would expect. For this reason it is suggested that at least two scaled and translated bump layers are applied to the plane to make the surface perturbations look much more natural (Figure 3.1).

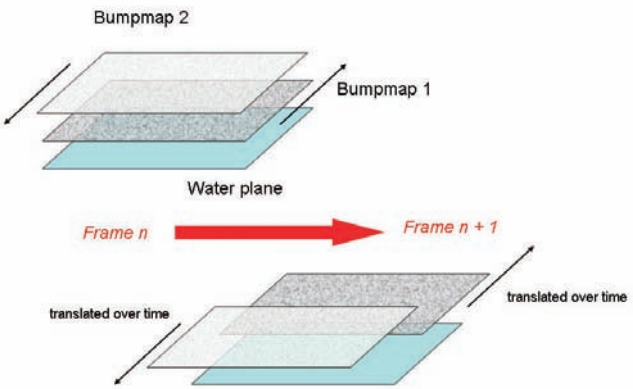


Figure 3.1. Bump map animation over two frames.

3.2.3 Reflection Render Pass

Reflection in the program is achieved through an additional render pass before the main render loop. The purpose of this is to render all of the geometry that needs to be reflected during the frame and store this information as a texture that can be applied to the water's surface during the main render. Before rendering the scene, the camera needs to be mirrored about the plane to give the correct view for the reflection.

To mirror the view matrix about the water plane, a custom transformation is required. The code in Listing 3.1 shows how this matrix is constructed (from `RenderReflectionTexture` in our example).

```
// Mirror the view matrix about the plane.
PVRTMat4 mMirrorCam(PVRTMat4::Identity());
mMirrorCam.ptr()[1] = -m_vPlaneWater.x;
mMirrorCam.ptr()[5] = -m_vPlaneWater.y;
mMirrorCam.ptr()[9] = -m_vPlaneWater.z;
mMirrorCam.ptr()[13] = -(2.0f * m_vPlaneWater.w);

m_mView = m_mView * mMirrorCam;
```

Listing 3.1. Constructing the view matrix.

As the diagram in Figure 3.2 shows, mirroring the camera is not enough by itself, because it results in the inclusion of objects below the water's surface, which spoils the reflection. This issue can be avoided by utilizing a user-defined clip plane along the surface of the water to remove all objects below the water from the render (See Section 3.3.1 for information on how this can be achieved in OpenGL ES 2.0). Using this inverted camera, the entire reflected scene can be rendered. Figure 3.3 shows the clipped reflection scene rendered to texture in the demo.

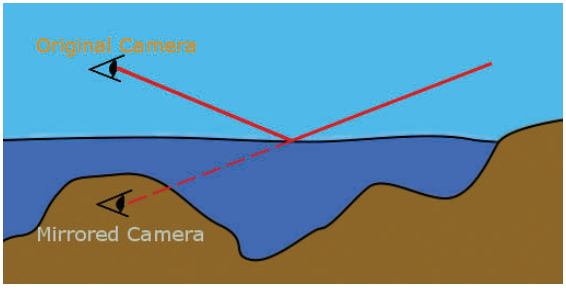


Figure 3.2. Mirrored camera for reflection.

Figure 3.3. Reflection stored as a texture.

In the main render pass, where the water plane is drawn, the reflected scene is sampled using screen-space coordinates, and then is distorted using an animated bump map normal.

The `gl_FragCoord` variable can be used to retrieve the current fragment's coordinate within the viewport, which is then normalized as follows:

```
myTexCoord = gl_FragCoord.xy * RcpWindowSize;

// where RcpWindowSize is a vec2 containing the reciprocal window
dimensions.
```

The multiply operation is used in place of a divide operation in order to reduce the required number of cycles.

To distort the reflection, an offset needs to be subtracted from these coordinates. This offset is calculated using a scaled normal sampled from a bump map. Figure 3.4 shows the perturbed result achieved from sampling the reflected scene using these distorted coordinates.

Used alone, this reflection technique results in unrealistically reflective water, since objects lying beneath the surface or coloring caused by dirt within the body of water are not considered. There are a number of steps that can be taken at this stage to improve the quality of the effect. The best option depends on the required aesthetics and performance of a particular implementation:

Figure 3.4. Water effect using only a permutated reflection texture.

- Perform another render pass to create a refraction texture (Section 3.2.4). This is expensive—reduced the performance of the effect on the development hardware by 60%, as was expected due to the extra pixel-render workload.
- Mix the color value of the fragment with a constant water color. This is good for simulating very murky water—reduced the performance by 9%, as expected due to the extra shader instructions per pixel.
- Alpha blend the water so objects below the water can be seen (Figure 3.5). This reduces the realism of the effect when rendering deep water because the edges of submerged objects will still appear sharp—reduced the performance by 11%, as expected due to the higher number of visible fragments per pixel.

It may be worth opting for one of the less intensive solutions if the required water effect is shallow, since there may be little value applying refraction in these cases. A mix between the texel color with a water color can be done in one of two ways, as discussed in Section 3.2.4.

Though introducing a new render pass comes at an additional cost, this can be reduced by adhering to the following recommendations:

1. *Render only what is necessary.* The CPU should be used to determine which objects are above the water plane and, if possible, which objects are

Figure 3.5. Water effect using a permutated reflection texture and alpha blending.

intersecting the plane since these are the only objects that will be needed during the render. If this still proves to be too expensive, the pass can be reduced to just drawing the key objects in the scene, such as a skybox and terrain.

2. *Favor FBO use over reading from the frame buffer.* Rather than using a copy function such as `glReadPixels()`, a frame buffer object with a texture bound to it should be used to store the output of the render pass in a texture [PowerVR 10]. This avoids the need to copy data from one memory location (the frame buffer) to another (texture memory), which would cost valuable cycles and bandwidth within the system. Even more important, it avoids direct access of the frame buffer that can result in a loss of parallelism, as the CPU would often be stalled, waiting for the GPU to render.

3. *Render to texture at the lowest acceptable resolution.* As the reflection texture is going to be distorted anyway, a lower resolution may be acceptable for it. A 256×256 texture has proven to be effective in the demo when running at a 640×480 display resolution, but depending on the maximum resolution of the screen on the platform being developed, this resolution could be reduced further. Keep in mind that a drop from 256×256 to 128×128 will result in a 75% lower resolution and workload.

4. *Avoid using* `discard` *to perform clipping.* Although using the discard keyword works for clipping techniques, its use decreases the efficiency of early order-independent depth rejection performance advantages that the POWERVR architecture offers (See Section 3.3.1 for more information).

3.2.4 Refraction Render Pass

In a case where the rendered water should appear to be fairly deep, adding refraction to the simulation can vastly improve the quality of the effect. To do this, an approach similar to that taken during the reflection render pass should be used, in which all objects below the water are rendered out to a texture (Figure 3.6). Clipping (using the inverse of the water plane) can be assisted by rough culling on the CPU beforehand. This reduces the GPU workload [PowerVR 10].

If the effect should produce very clear water, all elements of the scene below the water should be rendered, including the skybox (or similar object) (Figure 3.7). If a murky water effect is required, a fogging effect can be used to fade out objects at lower depths (discussed later in this section).

Once the scene has been rendered to a texture, it can then be utilized by the water's fragment shader. The screen-space texture coordinates (as used in Section 3.2.3) are also used to sample the refraction texture. The refraction sample is then combined with the reflection sample, either at a constant ratio (e.g., 50/50), or using an equation such as the Fresnel term to provide a dynamic

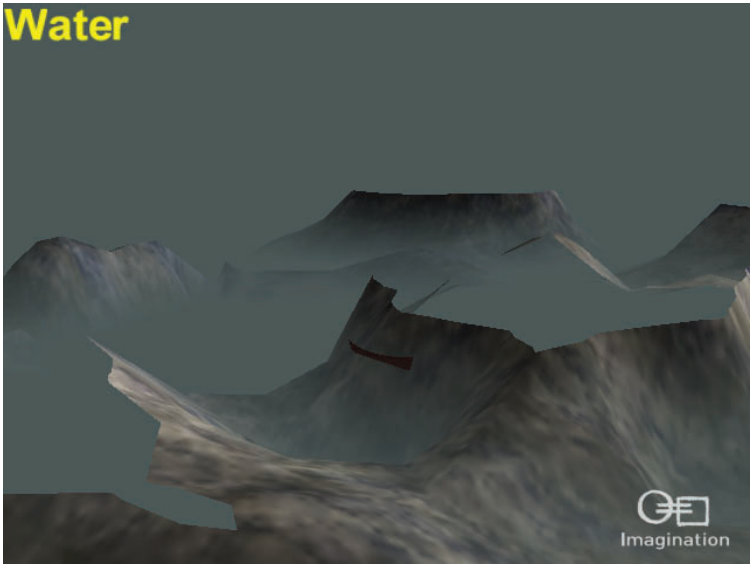


Figure 3.6. Refraction stored as a texture.

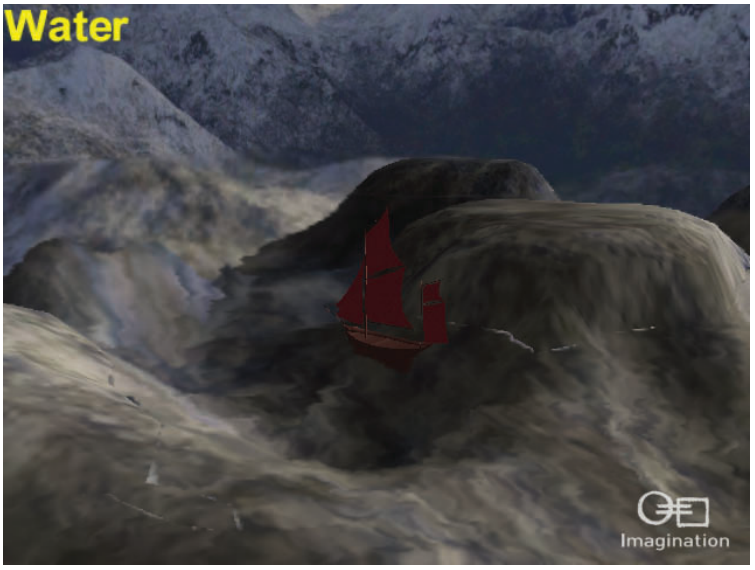


Figure 3.7. Water effect using only the refraction texture (without fogging).

Figure 3.8. Full water effect, showing edge artifact.

mixing of color, based on the current viewing angle. Figure 3.8 shows the full effect of the water, where reflection and refraction textures are mixed using the Fresnel term.

Fogging. A fogging effect to simulate water depth can be accomplished by setting a maximum fogging depth (below which objects are no longer visible) and using the value to perform a linear fade between the object's original color and the fog color based on the object's depth. To do this, the refraction pass must use the fog color as the clearing color; objects such as the skybox that should appear infinite, and therefore past the maximum fogging distance, should be omitted from the render (see Figure 3.9). Alternatively, a different fade can be used to create a more realistic fogging effect, such as an exponential fade, but this may come at an additional cost.

As the demo assumes the water plane's normal will always lie along the positive y-axis, the w-component of the water plane equation can be negated and passed into the shader as the water height (displacement from the center of the world space along the y-axis). The depth value of the vertex is then calculated in the vertex shader as an offset from this value:

```
vtx_depth=w-inVertex.y
```

The fragment shader then calculates the mixing ratio of the fog color and the object's color. This ratio is obtained using the following equation:

```
Ratio=(vtx_depth)/(max_fog_depth)
```

Figure 3.9. Water effect using only the refraction texture (with fogging).

(In practice, cycles can be saved by multiplying by the reciprocal of `max_fog_depth` instead of performing a division per vertex). This ratio is clamped to the range $[0, 1]$ and then is used to determine how much of the fogging color is applied to the fragment.

On the development hardware, disabling the fogging effect gave a 3.5% increase in performance level.

The `max_fog_depth` value can also be used on the CPU to cull geometry below this depth. This is a good way to decrease the number of objects that need to be rendered during the refraction pass. As with the reflection pass, this rough culling reduces the amount of geometry submitted, thus saving bandwidth and unnecessary computations.

Fresnel term. The Fresnel term is used to determine how much light is reflected at the boundaries of two semitransparent materials (the rest of which is absorbed through refraction into the second material). The strongest reflection occurs when the angle of incidence of the light ray is large, and, conversely, reflection decreases as the angle of incidence decreases (Figures 3.10 and 3.11). The Fresnel term provides a ratio of transmitted-to-reflected light for a given incident light ray.

In practice, this is used to determine the correct mix between the reflected and refracted textures for any point on the water's surface from the current view position. This is the Fresnel principle in reverse, and the ratio can be obtained using an approximation derived from the same equations.

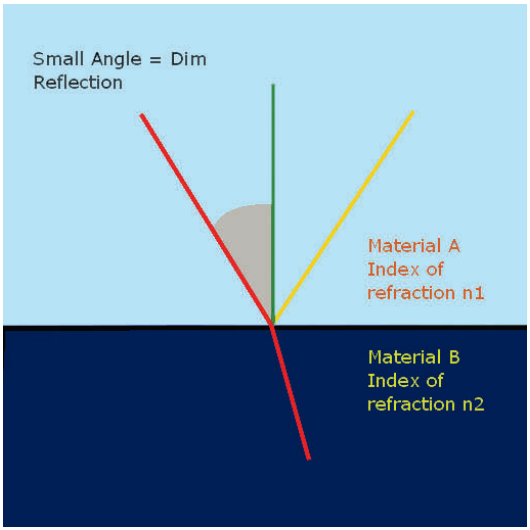


Figure 3.10. Small angle of Fresnel reflection.

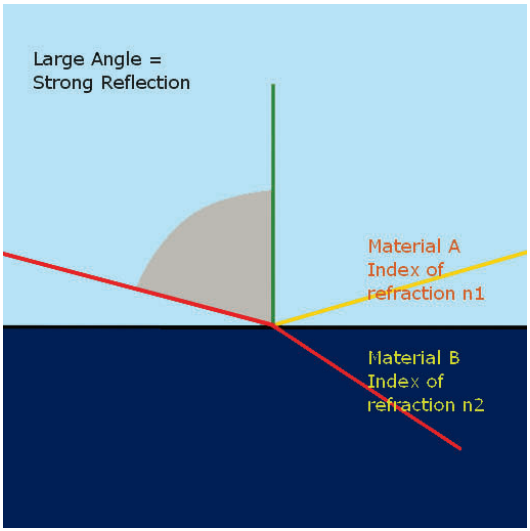


Figure 3.11. Large angle of Fresnel reflection.

The approximation of the Fresnel term used in the demo is determined using the following formulae, where n_1 and n_2 are the indices of refraction for each material [Pelzer 04]:

$$R(0) = \frac{(n_1 - n_2)^2}{(n_1 + n_2)^2},$$

$$R(\alpha) = (1 - R(0))(1 - \cos\alpha)^5 + R(0).$$

To save computation time, the result of the equation above is calculated outside of the application, using the values in Table 3.1.

Using these numbers, the constant terms in the formula can be precalculated (see Table 3.2).

n_1 (Air)	1.000293
n_2 (Water at room temperature)	1.333333

Table 3.1. Indices of refraction used in Fresnel term.

	Fresnel approximation
R(0)	0.02037
1–R(0)	0.97963

Table 3.2. Approximate values in Fresnel calculation.

The shader code in Listing 3.2 shows this principle in practice; first calculate the angle between the water normal (obtained from the bump map technique) and then the water-to-eye vector. These results are then used to calculate the Fresnel term, which is in turn used to mix the samples of the reflected and refracted scenes.

The normalization cube map used in the code is an optimization discussed later in Section 3.3.1. On some hardware this may achieve faster results than using the built-in `normalize()` functionality. The water normal here is assumed to be already normalized, though this may not always be the case.

Using the Fresnel calculation instead of a constant mix on the development hardware reduces the performance by 22%, but gives a much more realistic output.

```
// Use normalization cube map instead of normalize()
lowp vec3 vWaterToEyeCube =
    textureCube(NormalisationCubeMap,WaterToEye).rgb
        * 2.0 - 1.0;

mediump float fEyeToNormalAngle =
        clamp(dot(vWaterToEyeCube, vAccumulatedNormal), 0.0, 1.0);

// Use the approximations:
// R(0)-1 $\sim $= 0.98
// R(0) $\sim $= 0.02
mediump float fAirWaterFresnel = 1.0 - fEyeToNormalAngle;
fAirWaterFresnel = pow(fAirWaterFresnel, 5.0);
fAirWaterFresnel = (0.98 * fAirWaterFresnel) + 0.02;

// Blend reflection and refraction
lowp float fTemp = fAirWaterFresnel;
gl_FragColor = mix(vRefractionColour, vReflectionColour, fTemp);
```

Listing 3.2. Fresnel mix implementation.

3.3 Optimizations

3.3.1 User Defined Clip Planes in OpenGL ES 2.0

Although the programmability of the OpenGL ES 2.0 pipeline provides the flexibility to implement this water effect, there is a drawback in that the API does not have user-defined clip plane support, which is required to produce good quality reflections and refractions. Many OpenGL ES 2.0 text books suggest performing

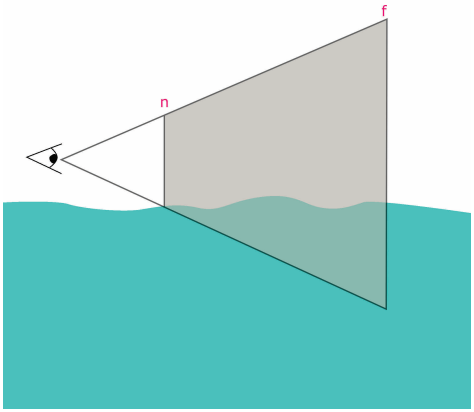


Figure 3.12. View frustum with n and f planes from original projection.

a render pass that uses a discard in the fragment shader so that fragments beyond the user-defined clip plane will be ignored. Although this method works and will produce the required output, using the `discard` keyword is highly inadvisable because it means the hardware is unable to perform early depth testing, and instead is forced to perform the full fragment shader pass. This cancels out specific performance advantages offered by some GPUs, such as those using early Z mechanisms or tile-based deferred rendering (TBDR) which include the POWERVR SGX architecture [PowerVR 10].

To solve this problem, a projection matrix modifying technique can be used [Lengyel 04]. The projection matrix (M) is used to convert all of the objects in the scene from view-space coordinates into normalized device coordinates (NDC), and part of this process is to clip objects that do not fall between the near, far, left, right, top, and bottom planes of the view frustum (Figure 3.12). By considering the function of the projection matrix in this way, it becomes apparent that there is already a built-in mechanism for clipping. Clipping along a user-defined plane can be achieved by altering the contents of the projection matrix, but this does introduce a number of problems (which will be discussed later).

The first stage of this technique requires the user-defined clip plane ($\vec{P}$) to be converted into view space. This can be done by multiplying the row vector representing the plane's coefficients (expressed in world space) by the inverse of the view matrix:

$$\vec{C} = \vec{P} \times M_{\text{view}}^{-1} = [C_x \quad C_y \quad C_z \quad C_w].$$

For this technique to work, the clipping plane must be facing away from the camera, which requires the C_w component to be negative. This does restrict the flexibility of the clipping method, but does not pose a problem for the clipping required for the water effect.

Altering the clipping planes requires operations on the rows of the projection matrix, which can be defined as

$$M = \begin{bmatrix} \vec{R}_1 \\ \vec{R}_2 \\ \vec{R}_3 \\ \vec{R}_4 \end{bmatrix}.$$

The near clipping plane ($\vec{n}$) is defined from the projection matrix Mas the third row plus the fourth row, so these are the values that need to be altered:.

$$\vec{n} = \vec{R}_3 + \vec{R}_4.$$

For perspective correction to work, the fourth row must keep the values $(0, 0, -1, 0)$. For this reason, the third row has to be

$$\vec{R}_3 = [C_x \quad C_y \quad C_z + 1 \quad C_w].$$

On the other hand, the far plane ($\vec{f}$) is calculated using the projection matrix by subtracting the third row from the fourth

$$\vec{f} = \vec{R}_4 - \vec{R}_3.$$

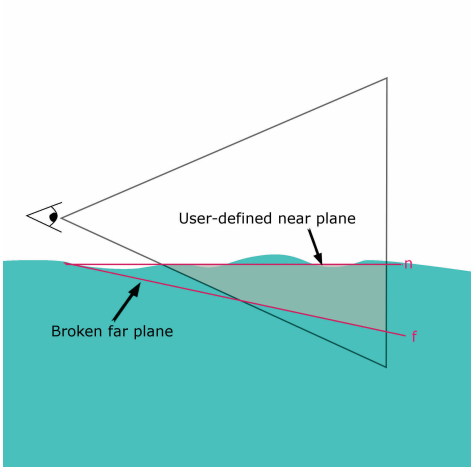


Figure 3.13. Modified view frustum with user-defined n and broken f.

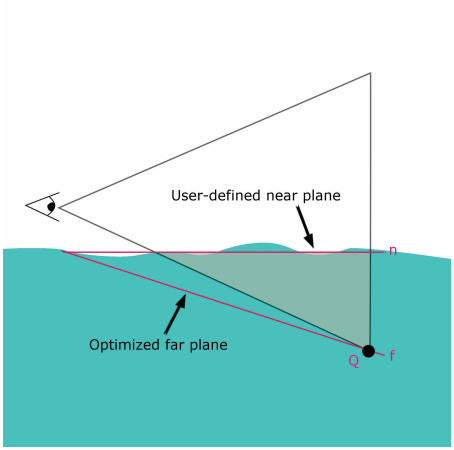


Figure 3.14. Corrected view frustum with user-defined n and optimized f.

Unfortunately, changing the near plane from its default direction along the positive z-axis results in a skewed far plane that no longer remains parallel with the near plane (Figure 3.13). This is due to the way in which the far plane is calculated in the above formula.

Although this problem cannot be corrected completely, the effect can be minimized by scaling the clip plane before the third row is set, which causes the orientation of the far clipping plane to change. Ideally, this scaling should result in an optimized far plane that produces the smallest possible view frustum that can still encapsulate the conventional view frustum (Figure 3.14).

To do this, the point ($\vec{Q}$) that lies furthest opposite the near plane within NDC must be calculated, using the following equation:

$$\vec{Q} = M^{-1} \begin{bmatrix} \operatorname{sgn}(C_x) & \operatorname{sgn}(C_y) & 1 & 1 \end{bmatrix}.$$

The result of this calculation can then be used to determine the scaling factor (a) that should be applied to the camera-space clip plane before it is used to alter the projection matrix:

$$a = \frac{2\vec{R}_4 \cdot \vec{Q}}{\vec{C} \cdot \vec{Q}}.$$

The camera-space plane can now be scaled before it is used to alter the projection matrix, using the following calculation:

$$\vec{C} = a\vec{C}.$$

Although this technique may seem more difficult to understand than the discard method of clipping, it is significantly faster because it allows the graphics hardware to perform clipping at almost no additional cost.

3.3.2 #define in GLSL

During production of this demo, it became apparent that using Booleans for if and if-else statements in vertex and fragment shaders resulted in at least an extra cycle for each decision point. This quickly became an issue because simple shaders that were covering large portions of the screen, such as the skybox, were processing needless cycles. Rolling out the code into a number of nearly identical shader files would provide a way around this issue, but would cause a lot of code duplication. To allow single shader files to be used repeatedly for multiple code paths, the `#define` preprocessor directive was used with `#ifdef` and `#else` in shader files so that code paths could be specified during compilation-time instead of at runtime.

The SDK allows this when using the method `PVRTShaderLoadFromFile()` by providing inputs for the additional parameters `aszDefineArray` and `uiDefArraySize`, where `aszDefineArray` is a pointer to an array of strings holding defines the user wants to append and `uiDefArraySize` is the number of defines in the array. This method automatically provides the `#define` and new-line character that need to be appended to the source code for each `define`, so the user only needs to provide an array of strings for the names, e.g., `A_USEFUL_DEFINE`.

Once a number of shaders have been loaded in this way, the shader used for an object can be changed during runtime to a shader with a different code path. Although this method creates a shader for each decision path (which uses more memory, but would also have been done if the code was rolled out in different files), it allows the performance of shaders to be improved by removing redundant cycles.

3.3.3 Further Optimizations/Improvements

Normalization cube map. On some hardware, the vector `normalize()` operation is costly and a texture lookup may be cheaper. On such platforms, normalization on a three-dimensional vector can be performed using a lookup to a normalization cube map. For demonstration purposes, the method of generating the normal map has been left in the code of this example, but time spent initializing the application could be saved by loading a preexisting normalization map instead.

The theory behind this method is simple; take a directional vector as the texture coordinate of the lookup and return the color at that position. This color represents the normalized value of any directional vector that points to its position. As the value retrieved using this technique is in texture space $[0, 1]$, a conversion into normal-space $[-1, 1]$ is required. On the development platform, this method approximately halved the number of cycles required to normalize a vector.

Scale water distortion. Without scaling the amount of distortion that is applied to each fragment, water in the distance can ultimately sample the reflection and

refraction textures at too large an offset, which gives water in the distance an unrealistic degree of distortion. Additionally, the bigger offset for distant fragments results in a higher amount of texture-read cache misses.

By scaling the amount of distortion that is applied to a given fragment, the visual quality of the effect can be improved and the number of stall cycles caused by texture cache misses can be reduced. This is done in the demo by dividing the wave's distortion value by the distance between the camera and the fragment's position (so fragments further from the camera are distorted less). The extra cycle cost has a minimal impact on performance (less than 1% on the test hardware) because, even though the texture-read stalls are reduced, they still account for the main bottleneck.

Render the water effect to a texture. Because of the heavy use of the fragment shader to produce the effect, the demo tends to be fragment limited on most hardware. To reduce this bottleneck, the water effect can be rendered to a texture at a lower resolution and then applied to the water plane during the final render pass. This technique benefits the speed of the demonstration by reducing the number of fragments that are rendered using the water effect. This can be further reduced (especially on a TBDR) by rendering objects that will obscure areas of the water in the final render pass, such as the demo's terrain. Although the introduction of numerous objects to the render can improve the speed of the water effect, the inaccuracies caused by mapping the texture to the final water plane can result in artifacts around the edges of models that were used during the low-resolution pass. Such artifacts are generally not that noticeable, provided that the shaders used for the additional objects in the low-resolution pass are the same as those used in the final render (i.e., rendering geometry without lighting during the low-resolution pass will cause highlights around dark edges of models in the final pass, so this should be avoided). One of the best ways to steer clear of the problems caused by the scaling is to avoid drawing objects that are very detailed around their edges that overlap the water because this reduces the likelihood of artifacts occurring. In the demo, the boat is omitted from the water's render pass because it is too detailed to be rendered without causing artifacts and does not afford as great a benefit as the terrain when covering areas of the water.

When rendering to a texture at a 256×256 resolution and performing the final render pass to a 640×480 screen, the reduction in quality is only slightly noticeable, but on the test hardware the performance level is increased by $\sim$18%.

Removing artifacts at the water's edge. One of the biggest problems with shader effects that perturb texture coordinates is the lack of control over the end texel that is chosen. Due to the clipping that is implemented in the reflection and refraction render passes, it is very easy for artifacts to appear along the edges of objects intersecting the water plane. The occurrence of artifacts occurs when

Figure 3.15. Full effect using artifact fix.

the sampled texel is taken from behind the object intersecting the water, which results in the texture sample being either the clear color, or geometry that ought to be obscured, resulting in visible seams near the water's edge. The edge artifact can be seen in Figure 3.8. To compensate, the clip-plane location is set slightly above the water surface (a small offset along the positive y-axis). In the case of the refraction render pass, such an offset will cause some of the geometry above the water to be included in the rendered image, which helps to hide the visible seams by sampling from this above-the-water geometry.

Although another inaccuracy is introduced because of the deliberately imperfect clipping, it is barely noticeable, and the effect of the original artifact is effectively removed for very little additional computation. The same benefit applies to the reflected scene, although in this case the offset direction is reversed, and clipping occurs slightly below the water. Figure 3.15 shows the scene with the artifact fix in place.

Another way to compensate for the artifacts, and improve the aesthetics of the effect, is to use fins or particle effects along the edges of objects intersecting the water to give the appearance of a wake where the water is colliding with the objects. The drawback of these techniques is that they both require the program to know where in the scene objects are intersecting the water, which can be very expensive if the water height is changing or objects in the water are moving dynamically.

3.4 Conclusion

We have presented a technique that allows a water effect to be augmented onto a simple plane, using several render passes, some simple distortion, and texture mixing in the fragment shader. Additionally, we have presented optimal techniques for user-defined clip planes, normalization, removing artifacts caused by texture-coordinate perturbation, and have also highlighted the benefits of utilizing low-resolution render passes to reduce the fragment shader workload. The result is a high-performance example with extremely compelling visual results. Though this example targets the current low-cost OpenGL ES 2.0 capable devices, it can be correspondingly scaled to take advantage of higher resolution displays and increased GPU power.

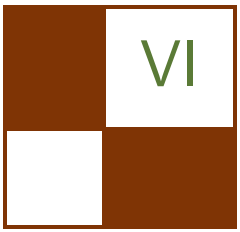
Bibliography

[PowerVR 10] Imagination Technologies. "POWERVR SGX OpenGL ES 2.0 Application Development Recommendations." www.imgtec.com/.../POWERVR%20SGX.OpenGL%20ES%202.0%20Application%20Development%20Recommendations, 2010.

[Feneny 03] Simon Fenney, "Texture Compression using Low-Frequency Signal Modulation." In *Proceedings Graphics Hardware*, pp. 84–91. New York: ACM, 2003.

[Lengyel 04] Eric Lengyel. "Modifying the Projection Matrix to Perform Oblique Near-plane Clipping." Terathon Software 3D Graphics Library, 2004. Available at http://www.terathon.com/code/oblique.html.

[Pelzer 04] Kurt Pelzer. "Advanced Water Effects." In *ShaderX*^{2}. Plano, TX: Wordware Publishing, Inc, 2004.



VI 3D Engine Design

In this part, we focus on engine-level optimization techniques useful for modern games with large and open worlds.

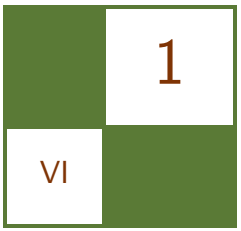
We start with Stephen Hill and Daniel Collin's article "Practical, Dynamic Visibility for Games," in which the authors introduce methods for determining per-object visibility, taking into account occlusion by other objects. The article provides invaluable and inspiring experience from published AAA titles, showing excellent gains that are otherwise lost without this system.

Next, Eric Penner presents "Shader Amortization using Pixel Quad Message Passing." In this chapter, Eric analyzes one particular aspect of modern programmable hardware: the pixel derivative instructions and pixel quad rasterization. The article identifies a new level at which optimizations can be performed, and applies this method to achieve results such as 4×4 percentage closer filtering (PCF) using only one texture fetch, and 2×2 bilateral up-sampling using only one or two texture fetches.

Finally, on the topic of crowd rendering, the chapter "A Rendering Pipeline for Real-Time Crowds," by Benjamin Hernandez and Isaac Rudomin, describes a detailed system for simulating and rendering large numbers of different characters on the GPU, making use of optimizations such as culling and LOD-selection to improve performance of the system.

I hope you find these articles inspiring and enlightening for your rendering and engine development work.

—Wessam Bahnassi



Practical, Dynamic Visibility for Games

Stephen Hill and Daniel Collin

1.1 Introduction

With the complexity and interactivity of game worlds on the rise, the need for efficient dynamic visibility is becoming increasingly important.

This article covers two complementary approaches to visibility determination that have shipped in recent AAA titles across Xbox 360, PS3, and PC: *Splinter Cell Conviction* and *Battlefield: Bad Company 1 & 2*.

These solutions should be of broad interest, since they are capable of handling completely dynamic environments consisting of a large number of objects, with low overhead, straightforward implementations, and only a modest impact on asset authoring.

Before we describe our approaches in detail, it is important to understand what motivated their development, through the lens of existing techniques that are more commonly employed in games.

1.2 Surveying the Field

Static potentially visible sets (PVSs) is an approach popularized by the Quake engine [Abrash 96] and is still in common use today, in part because of its low runtime cost. Put simply, the world is discretized in some way (BSP, grid, etc.) and the binary visibility from each sector (leaf node, cell, or cluster, respectively) to all other sectors is precomputed and stored. At runtime, given the current sector containing the camera, determining the set of potentially visible objects becomes a simple matter of retrieving the potentially visible sectors (and by extension, their associated objects) and performing frustum culling.

One major drawback of using PVS by itself is that any destructible or moving objects (e.g., doors) typically have to be treated as nonoccluding from the perspective of visibility determination. This naturally produces over inclusion—in

addition to that coming from sector-to-sector visibility—and can therefore constrain level-design choices in order to avoid pathological situations.

Another disadvantage stems from the fact that a PVS database can be extremely time consuming to precompute,[1] which may in turn disrupt or slow production.

Portals are another approach that can complement or replace static PVS. Here, sectors are connected via convex openings or "portals" and the view frustum is progressively clipped against them [Akenine-Möller et al. 08], while objects are simultaneously gathered and tested against the active subfrustum.

Since clipping happens at runtime, the state of portals can be modified to handle a subset of dynamic changes to the world, such as a door closing or opening. But, even though portals can ameliorate some of the limitations of a static PVS solution, they are still best suited to indoor environments, with corridors, windows, and doorways providing natural opportunities to constrain and clip the view frustum.

Antiportals are a related technique for handling localized or dynamic occlusion whereby, instead of constraining visibility, convex shapes are used to occlude (cull away) objects behind them with respect to the player. Though antiportals can be effective in open areas, one can employ only a limited number in any given frame, for performance reasons. Similarly, occluder fusion—culling from the combined effect of several antiportals—is typically not viable, due to the much higher cost of inclusion testing against concave volumes.

In recent years, hardware occlusion queries (OQs) have become another popular tool for visibility determination [Soininen 08]. The canonical approach involves rendering the depth of a subset (or a simplified representation) of the scene—the occluders—and then rasterizing (without depth writes) the bounds of objects, or groups of objects. The associated draw calls are bracketed by a query, which instructs the GPU to count pixels passing the depth test. If a query returns that no pixels passed, then those objects can be skipped in subsequent rendering passes for that camera.

This technique has several advantages over those previously discussed: it is applicable to a wider range of environments, it trivially adapts to changes in the world (occluders can even deform), and it handles occluder fusion effortlessly, by nature of z-buffer-based testing. In contrast, whereas both static PVS and portals can handle dynamic objects, too, via sector relocation, those objects cannot themselves occlude in general.

1.3 Query Quandaries

On paper OQs are an attractive approach, but personal experience has uncovered a number of severe drawbacks, which render them unsuitable for the afore-

[1] On the order of 10 hours, in some cases [Hastings 07].

mentioned titles. We will now outline the problems encountered with occlusion queries.

1.3.1 Batching

First, though OQs can be batched in the sense that more than one can be issued at a time [Soininen 08]—thereby avoiding lock-step CPU-GPU synchronization—one cannot batch several bounds into a single draw call with individual query counters. This is a pity, since CPU overhead alone can limit the number of tests to several hundred per frame on current-generation consoles, which may be fine if OQs are used to supplement another visibility approach [Hastings 07], but is less than ideal otherwise.

1.3.2 Latency

To overcome latency, and as a general means of scaling OQs up to large environments, a hierarchy can be employed [Bittner et al. 09]. By grouping, via a bounding volume hierarchy (BVH) or octree for instance, tests can be performed progressively, based on parent results, with sets of objects typically rejected earlier.

However, this dependency chain generally implies more CPU-GPU synchronization within a frame since, at the time of this writing, only the CPU can issue queries.[2] Hiding latency perfectly in this instance can be tricky and may require overlapping *query* and *real* rendering work, which implies redundant state changes in addition to a more complicated renderer design.

1.3.3 Popping

By compromising on correctness, one can opt instead to defer checking the results of OQs until the next frame—so called latent queries [Soininen 08]—which practically eliminates synchronization penalties, while avoiding the potential added burden of interleaved rendering. Unfortunately, the major downside of this strategy is that it typically leads to objects "popping" due to incorrect visibility classification [Soininen 08]. Figure 1.1 shows two cases where this can occur. First, the camera tracks back to reveal object A in Frame 1, but A was classified as outside of the frustum in Frame 0. Second, object B moves out from behind an occluder in Frame 1 but was previously occluded in Frame 0.

Such artifacts can be reduced by extruding object-bounding volumes,[3] similarly padding the view frustum, or even eroding occluders. However, these fixes come with their own processing overhead, which can make eliminating all sources of artifacts practically impossible.

[2] Predicated rendering is one indirect and limited alternative on Xbox 360.

[3] A more accurate extrusion should take into account rotational as well as spatial velocity, as with continuous collision detection [Redon et al. 02].

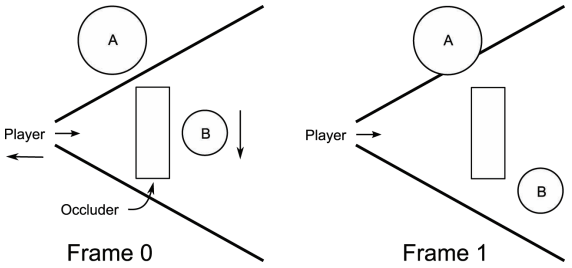


Figure 1.1. Camera or object movement can lead to popping with latent queries.

Sudden changes such as camera cuts are also problematic with latent queries, potentially leading to either a visibility or processing spike [Hastings 07] to avoid rampant popping. As such, it may be preferable to simply skip rendering for a frame and only process visibility updates behind the scenes.

1.3.4 GPU Overhead

The GPU is a precious resource and a common bottleneck in games, as we seek to maximize visual fidelity for a given target frame rate. Therefore, a visibility solution that relies heavily on GPU processing is less than ideal, particularly on modern consoles where multiple CPU cores or SPUs are available. While it is true to say that OQs should be issued only when there is an overall GPU saving [Soininen 08], this cannot be guaranteed in general and we would ideally like to dedicate as much GPU muscle as possible to direct rendering.

1.3.5 Variable Costs

A final disadvantage with OQs is that the cost of testing an object is roughly proportional to its size on screen, which typically does not reflect its true rendering cost. While one can, for instance, choose to always render objects with high screen-space coverage to avoid this penalty, it is a less viable strategy when working with a hierarchy.

Even if one develops a more sophisticated oracle [Bittner et al. 09] to normalize performance, this can come at the cost of reduced culling effectiveness. Furthermore, a hierarchy requires additional CPU overhead when objects move, or parts become visible or occluded. As with queries, the per-frame update cost can be bounded by distributing work over multiple frames, but this can similarly compromise culling.

Ideally we would like to avoid these kinds of unfortunate trade-offs, especially when major changes to the environment occur; although leveraging coherency can be a great way to reduce the average per-frame processing time, it should not exacerbate worst-case performance!

1.4 Wish List

Ideally we would like to take the strengths of OQs but reduce or eliminate the negatives. Here is our wish list:

These first items are already taken into account with OQs:

- no precomputation
- general applicability
- cccluder fusion

Here is a list of improvements we would like to achieve:

- low latency
- reduced CPU-GPU dependency
- no reliance on coherency
- bounded, high performance
- simple, unified solution

In summary, we would like to be able to handle a wide range of dynamic scenes with the minimum of fuss and no less than great performance. Essentially, we *want it all* and in the case of *Splinter Cell Conviction*—as you will now learn—we wanted it *yesterday*!

1.5 *Conviction* Solution

One of the initial technical goals of *Splinter Cell Conviction* was to support dense environments with plenty of clutter and where, in some situations, only localized occlusion could be exploited.

We initially switched from PVS visibility to OQs because of these requirements, but having battled for a long time with the drawbacks outlined earlier, and becoming increasingly frustrated by mounting implementation complexity, hacks, and failed work-arounds, we started to look for alternatives. Unfortunately, by this point we had little time and few resources to dedicate to switching solutions yet again.

Luckily for us, [Shopf et al. 08] provided a guiding light, by demonstrating that the hierarchical Z-buffer (HZB) [Greene et al. 93] could be implemented efficiently on modern GPUs—albeit via DX10—as part of an AMD demo. The demo largely validated that the HZB was a viable option for games, whereas we had previously been skeptical, even with a previous proof of concept by [Décoret 05].

Most importantly, it immediately addressed all of our requirements, particularly with respect to implementation simplicity and bounded performance. In

fact, the elegance of this approach cannot be understated, comparing favorably with the illusory simplicity of OQs, but without any of the associated limitations or management complexity in practice.

1.5.1 The Process

The steps of the process are detailed here.

Render occluder depth. As with OQs, we first render the depth of a subset of the scene, this time to a render target texture, which will later be used for visibility testing, but in a slightly different way than before.

For *Conviction*, these occluders were typically artist authored[4] for performance reasons, although any object could be optionally flagged as an occluder by an artist.

Create a depth hierarchy. The resulting depth buffer is then used to create a depth hierarchy or z-pyramid, as in [Greene et al. 93]. This step is analogous to generating a mipmap chain for a texture, but instead of successive, weighted down-sampling from each level to the next, we take the maximum depth of sets of four texels to form each new texel, as in Figure 1.2.

This step also takes place on the GPU, as a series of quad passes, reading from one level and writing to the next. To simplify the process, we restrict the visibility resolution to a power of two, in order to avoid the additional logic of [Shopf et al. 08]. Figure 1.3 shows an example HZB generated in this way.

In practice, we render at 512×256,[5] since this seems to strike a good balance between accuracy and speed. This could theoretically result in false occlusion for objects of 2×2 pixels or less at native resolution, but since we contribution-cull small objects anyway, this has not proven to be a problem for us.

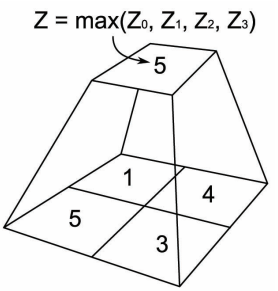


Figure 1.2. Generating successive levels of the HZB.

[4] These are often a simplified version of the union of several adjoining, structural meshes.
[5] This is approximately a quarter of the resolution of our main camera in single-player mode.

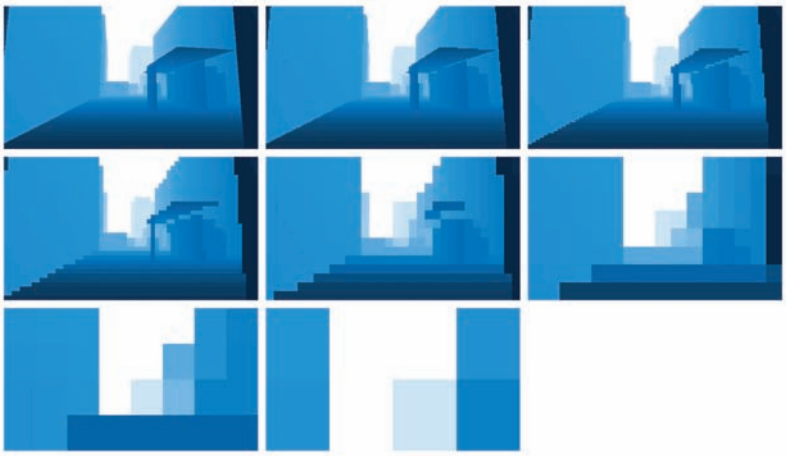


Figure 1.3. The resulting depth hierarchy. Note that the sky in the distance increasingly dominates at coarser levels.

Test object bounds. We pack object bounds (world-space AABBs) into a dynamic point-list vertex buffer and issue the tests as a single draw call. For each point, we determine, in the vertex shader, the screen-space extents of the object by transforming and projecting the bounds (see Figure 1.4). From this, we calculate the finest mip level of the hierarchy that covers these extents with a fixed number of texels or fewer and also the minimum, projected depth of the object (see Listing 1.1).

```
//Contains the dimensions of the viewport.
//In this case x = 512, y = 256
float2 cViewport;

OUTPUT main(INPUT input)
{
    OUTPUT output;

    bool visible = !FrustumCull(input.center, input.extents);

    // Transform/project AABB to screen-space
    float min_z;
    float4 sbox;
    GetScreenBounds(input.center, input.extents, min_z, sbox);

    // Calculate HZB level
    float4 sbox_vp = sbox*cViewport.xyxy;
    float2 size = sbox_vp.zw - sbox_vp.xy;
    float level = ceil(log2(max(size.x, size.y)));
```

```
    output.pos = input.pos;
    output.sbox = sbox;
    output.data = float4(level, min_z, visible, 0);

    return output;
}
```

Listing 1.1. HZB query vertex shader.

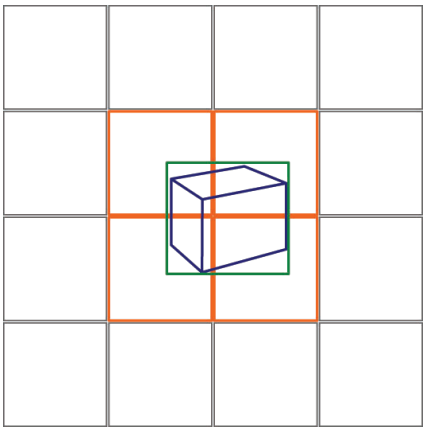


Figure 1.4. The object's world-space AABB (blue), screen extents (green) and overlapping HZB texels (orange).

This depth, plus the UVs (sbox is the screen-space AABB) and mip level for HZB lookup are then passed to the pixel shader. Here we test for visibility by comparing the depth against the overlapping HZB texels and write out 1 or 0 as appropriate (see Listing 1.2).

```
sampler2D sHZB : register(s0);

float4 main(INPUT input) : COLOR0
{
    float4 sbox = input.sbox;
    float level = input.data.x;
    float min_z = input.data.y;
    bool visible = input.data.z;

    float4 samples;
    samples.x = tex2Dlod(sHZB, float4(sbox.xy, 0, level)).x;
    samples.y = tex2Dlod(sHZB, float4(sbox.zy, 0, level)).x;
```

```
    samples.z = tex2Dlod(sHZB, float4(sbox.xw, 0, level)).x;
    samples.w = tex2Dlod(sHZB, float4(sbox.zw, 0, level)).x;

    float max_z = max4(samples);

    // Modulate culling with depth test result
    visible *= min_z <= max_z;

    return visible;
}
```

Listing 1.2. HZB query pixel shader.

In practice, we chose to use 4×4 HZB depth comparisons in contrast to the simpler example code above, since this balanced ALU instructions and texture lookups on the Xbox 360.

Also note that we perform world-space frustum testing and screen-bound generation separately. While the two can be combined as in [Blinn 96], we found that we got better code generation by performing them separately and could add additional planes to the frustum test when processing visibility for mirrors.

It is also possible to improve on the mip level selection for situations when an object covers fewer texels.

Process the results. Finally, the results are read back to the CPU via MemExport on Xbox 360. On PC, under DX9, we instead emulate DX10 stream-out by rendering with a point size of one to an off-screen render-target, followed by a copy to system memory via `GetRenderTargetData`.

1.5.2 Tradeoffs

By using a fixed number of lookups instead of rasterization, the performance of the visibility tests is highly predictable for a given number of objects. That said, this bounded performance comes at the cost of reduced accuracy for objects that are large on screen.

On the other hand, this approach can be viewed as probabilistic: large objects are, on average, more likely to be visible anyway, so performing more work (in the form of rasterization with OQs) is counter-productive. Instead, with HZB testing, accuracy is distributed proportionally. This proved to be a particularly good fit for us, given that we wanted a lot of relatively small clutter objects, for which instancing was not appropriate for various reasons.

We also benefited from the high granularity afforded by a query per object, whereas wholly OQ-based methods require some degree of aggregation in order to be efficient, leading to reduced accuracy and more variable performance. This became clear in our own analysis when we switched to HZB visibility from OQs. We started off with a 2×2 depth-test configuration, and even that out-performed

hand-placed occlusion query volumes, both in terms of performance and amount of culling. Essentially, what we lost in terms of occlusion accuracy, we gained back in being able to test objects individually.

Point rendering with a vertex buffer was chosen primarily for ease of development because vertex buffers offered the convenience of heterogeneous data structures. However, a more efficient option could be to render a single quad and fetch object information from one or more textures instead. Not only would this ensure better pixel-quad utilization on some hardware, but it would also play to the strength of GPUs with a nonunified shader architecture such as the PS3's RSX, where the bulk of the shader hardware is dedicated to pixel processing.

1.5.3 Performance

Table 1.1 represents typical numbers seen in PIX on Xbox 360, for a single camera with around 22000 objects, all of which are processed in each each frame.

Pass	Time (ms)
Occlusion	0.06
Resolve	0.04
HZB Generation	0.10
HZB Queries	0.32
Total	**0.52**

Table 1.1. Performance timings.

1.5.4 Extensions

Once you have a system like this in place, it becomes easy to piggy-back related work that could otherwise take up significant CPU time compared with the GPU, which barely breaks a sweat. Contribution fading/culling, texture streaming and LOD selection, for instance, can all be determined based on each object's screen extents,[6] with results returned in additional bits.

On Xbox 360, we can also bin objects into multiple tiles ourselves, thereby avoiding the added complexity and restrictions that come with using the predicated tiling API, not to mention the extra latency and memory overhead when double-buffering the command buffer.

Finally, there is no reason to limit visibility processing to meshes. We also test and cull lights, particle systems, ambient occlusion volumes [Hill 10], and dynamic decals.

[6]We choose to use the object's bounding sphere for rotational invariance.

1.5.5 Shadow Caster Culling

We also extend our system to accelerate shadow-map rendering, with a two-pass technique initially inspired by [Lloyd et al. 04], but with a more straightforward approach. For instance, we do not slice up the view frustum and test subregions as they do. This is primarily because we are not using shadow volumes for rendering and therefore are not aiming to minimize fill-rate[7]—only the number of casters—for CPU and vertex transform savings. Development time and ease of GPU implementation are also factors.

In the first pass, we test caster visibility from the light's point of view, in exactly the same way that we do for a regular camera: via another HZB. If a given caster is visible, we write out the active shaft bounds, which are formed from the 2D light-space extents, the caster's minimum depth, and the maximum depth from the HZB (see Listing 1.3), otherwise it is culled as before:

```
float3 shaft_min = float3(input.sbox.xy, min_z)
float3 shaft_max = float3(input.sbox.zw, max_z)
```

Figure 1.5 shows this in action for a parallel light source. Here, caster C is fully behind an occluder,[8] so it can be culled away since it will not contribute to the shadow map.

In the second pass, we transform these shafts into camera space and test their visibility from the player's point of view via the existing player camera HZB—again just like regular objects. Here, since the shafts of A and B have been clamped to the occluder underneath, they are not visible either.

```
// Use the lower level if we only touch <= 2 texels
// in both dimensions

float level_new = max(level - 1, 0);
float2 scale = pow(2, -level_new);

float2 a = floor(sbox_vp.xy*scale);
float2 b = ceil(sbox_vp.zw*scale);

float2 dims = b - a;

if (dims.x <= 2 && dims.y <= 2)
    level = level_new;
```

Listing 1.3. HZB level refinement.

[7]But we could adapt this type of testing to cull more. See Section 1.7.
[8]Occluders used for shadow culling always cast shadows.

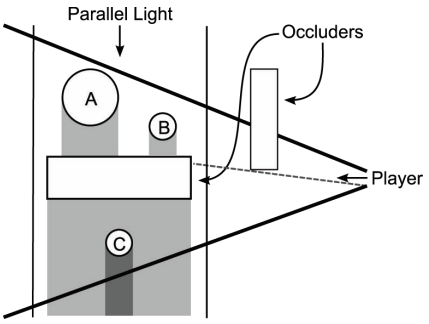


Figure 1.5. Two-pass shadow visibility.

Conceptually, we are exploiting the redundancy of shadow-volume overlap across two viewpoints in order to reduce our set of potential casters.

1.5.6 Summary

To reiterate, this entire process takes place as a series of GPU passes; the CPU is involved only in dispatching the draw calls and processing the results at the end.

In retrospect, a CPU solution could have also worked well as an alternative, but we found the small amount of extra GPU processing to be well within our budget. Additionally, we were able to leverage fixed-function rasterization hardware, stream processing, and a mature HLSL compiler, all with literally man-years of optimization effort behind them. In contrast to the simple shaders listed earlier, a hand-optimized VMX software rasterizer would have taken significantly longer to develop and would have been harder to extend.

If you already have a PVS or portal visibility system, there can still be significant benefits to performing HZB processing as an additional step. In the first place, either system can act as an initial high-level cull, thus reducing the number of HZB queries. In the case of portals, the "narrow-phase" subfrusta testing could also be shifted to the GPU. Indeed, from our own experience, moving basic frustum testing to the GPU alone was a significant performance improvement over VMX tests on the CPU. Finally, in the case of BSP-based PVS, the faces could be preconverted to a number of large-scale occluders for direct rendering.

1.6 *Battlefield* Solution

When developing the game (*Battlefield: Bad Company 1)* using our new in-house Frostbite engine for the first time, we knew that we needed a solution for removing objects occluded by others. We discussed many methods, but it all came down to a list of things that we wanted the system to have:

- must be fully dynamic, since the environment—both objects and terrain—can deform
- low GPU overhead
- results accessible from the CPU, so we can skip updating certain aspects of occluded objects, such as animation

After reading about Warhawk's approach [Woodard 07] based on software rasterization on SPUs, we decided to try a similar approach since we had spare processing power available on the CPU side. The resulting implementation was subsequently rolled out across all of our target platforms (PlayStation 3, Xbox 360, and PC), but we will focus in particular on the details of the PS3 version.

At a high level, the steps involved are very similar to those used for *Conviction*: a software occlusion rasterizer renders low polygon meshes to a z-buffer, against which occluders are tested to determine if they are visible or not.

In reality, the work is broken down into a number of stages, which are job-scheduled in turn across several SPUs. We will now describe these in detail.

1.6.1 The Process

Occluder triangle setup. This stage goes through all occluders in the world space (a flat array) in preparation for rasterization:

1. Each job grabs a mesh from the array using `InterlockedIncrement`.
2. The job checks if the mesh is inside the frustum. If it is not, it continues to the next one (Step 1).
3. If the mesh is fully inside the frustum, its triangles are immediately appended to an output array (also interlocked and shared between the jobs).
4. If the mesh was not fully inside, its triangles are clipped before being added.

Terrain triangle setup. This is effectively the same as the previous stage, except that it generates and adds conservative triangles for the terrain[9] to the array.

Occluder render. This is the stage that actually rasterizes the triangles. Each SPU job has its own z-buffer (256×114) and grabs 16 triangles at a time from the triangle array generated previously.

When the jobs are finished getting triangles from the triangle array, they will each try to lock a shared mutex. The first one will simply DMA its z-buffer to main memory, unlock the mutex, and exit so that the next job can start running.

As the mutex gets unlocked, the next job will now merge its own buffer with the one in main memory and send back the result, and so on. (*Note*: There are

[9] As the terrain can deform, these must be regenerated.

several ways to improve on this and make it faster. We could, for example, DMA directly from each SPU.)

Frustum cull. This stage performs frustum versus sphere/bounding box (BB) checks on all meshes in the world—typically between 10,000 and 15,000—and builds an array for the next stage. The implementation traverses a tree of spheres (prebuilt by our pipeline) and at each leaf we do bounding-box testing if the sphere is not fully inside.

Occlusion cull. Finally, this is where visibility testing against the z-buffer happens. We first project the bounding box of the mesh to screen-space and calculate its 2D area. If this is smaller than a certain value—determined on a per-mesh basis—it will be immediately discarded (i.e, contribution culled).

Then, for the actual test against the z-buffer, we take the minimum distance from the camera to the bounding box and compare it against the z-buffer over the whole screen-space rectangle. This falls somewhere between the approach of [Woodard 07]—which actually rasterizes occluders—and that of *Conviction* in terms of accuracy.

Performance The timings reflect best-case parallelism over five SPUs and were measured in a typical scene (see Table 1.2). In practice, workloads between SPU jobs will vary slightly and may be intermixed with other jobs, so the overall time for visibility processing will be higher in practice.

In this case we rasterized around 6000 occluder triangles (we normally observe 3000 to 5000), and performed around 3000 occlusion tests after frustum and extent culling.

Stage	**Time/SPU (ms)**
Triangle Setup	0.4
Rasterization	1.0
Frustum Cull	0.6
Occlusion Cull	0.3
Total	**2.3**

Table 1.2. Performance timings.

1.7 Future Development

1.7.1 Tools

Although artist-authored occluders are generally a good idea for performance reasons (particularly so with a software rasterizer), we encountered a couple of notable problems with this strategy on *Conviction*. First, with a large team and

therefore a number of people making changes to a particular map, there were a few cases where modifications to the layout of visual meshes would not be applied to the associated occluders. Even with the blueprint of a map largely locked down, cosmetic changes sometimes introduced significant errors and these tended to occur right at the end of testing when production was most stretched!

Second, some artists had a tendency to think of modeling occluders in the same way as collision meshes—when, in fact, occluders should always be flush with, or inside of, the visual meshes they represent—or they did not feel that a small inaccuracy would be that important. This simply was not the case: time and again, testers would uncover these problems, particularly in "scope mode" where the reduced field of view can magnify these subtle differences up to half of the screen, causing large chunks of the world to disappear.

These errors would also show up as "shadow acne" due to the requirement that shadow occluders—those used for culling casters during shadow map visibility—had to cast shadows themselves. Sometimes, it would have made more sense to have just used these visual meshes directly as occluders, instead of creating separate occluder meshes.

Though checks can be added in the editor to uncover a lot of these issues, another option could be to automatically weld together, simplify, and chunk up existing visual meshes flagged by artists.

At the root of it all, the primary concern is correctness; there is no such thing as "pretty looking" visibility, so one could argue that it is not the best use of an artist's time to be modeling occluders if we can generate them automatically for the most part, particularly if a human element can introduce errors. This is definitely something we would like to put to the test, going forward.

1.7.2 Optimizations

One trivial optimization for the GPU solution would be to add a pre-pass, testing a coarse subdivision of the scene (e.g., regular grid) to perform an earlier, high-level cull—just like in *Battlefield*, but using the occlusion system too. We chose not to do this since performance was already within our budget, but it would certainly allow the approach to scale up to larger environments (e.g., "open world").

Additionally, a less accurate *object-level* pre-pass (for instance, four HZB samples using the bounding sphere, as with [Shopf et al. 08]) could lead to a speed up wherever there is a reasonable amount of occlusion (which by necessity is a common case). Equally, a finer-grained final pass (e.g., 8×8 HZB samples) could improve culling of larger occluders.

In a similar vein, another easy win for the SPU version would be using a hierarchical z-buffer either for early rejection or as a replacement for a complete loop over the screen bounds. As earlier numbers showed, however, the main hotspot performance-wise is occluder rasterization. In that instance we might

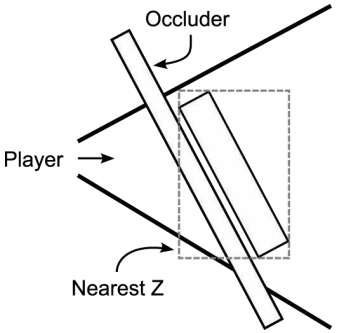


Figure 1.6. A single screen-space z-value for occluders can lead to conservative acceptance in some cases.

gain again, this time from *hierarchical* rasterization as in [Abrash 09], although at the cost of increased implementation complexity. Frustum culling could also be sped up by switching to a different data structure (e.g., grid) to improve load balancing on SPUs as well as memory access patterns.

Although the accuracy-performance trade-off from the HZB was almost always beneficial for *Conviction*, we did encounter a couple of instances where we could have profited from better culling of large, structural geometry. We believe that the biggest factor here was the lack of varying z over the occluder (see Figure 1.6) when testing against the HZB, not the number of tests (beyond 4×4) or the base resolution.

On Xbox 360, we investigated hardware-rasterizing occluder bounds as a proof-of-concept for overcoming this, but we ran out of time and there were some performance pitfalls with MemExport. We hope to pick up where we left off in the future.

Conviction's shadow-caster culling proved to be a significant optimization for cascaded shadow maps. One potential avenue of future development would be to try to adapt the idea of frustum subdivision coupled with caster-receiver intersection testing, as presented in [Diamand 10], with similarities to [Lloyd et al. 04]. [Eisemann and Décoret 06] and [Décoret 05] also build on the latter.

We would also like to extend culling to local shadow lights. As we already cache casters per shadow light (the cache is updated on object or light movement), we could directly evaluate shadow visibility for this subset of the scene. This would avoid the higher fixed overhead of processing all objects in the map as we do for the main view or shadow cascades, which is important since we can have up to eight active shadow lights per camera. These updates could happen either every frame or whenever the list changes.

1.7.3 Future Hardware

The jury is out on exactly what sort of future we face when it comes to the convergence of increasingly multi-core CPUs and more programmable GPUs and when, or indeed if, it will happen. Larrabee is an interesting example, showing that even fixed-function rasterization hardware is potentially on the way out [Abrash 09] and, while a CPU solution could be considered a safe long-term bet, the most efficient method going forward may be closer to the way hardware works than a traditional scan-line approach.

[Andersson 10] describes two possible future scenarios for visibility processing: either a progression of the GPU approach we already described, but with lower latency, or having the ability for the GPU to feed itself commands. A killer application for the latter could be shadow-map rendering, where visibility (as earlier) and subsequent draw calls would happen entirely on the GPU, thereby avoiding any CPU synchronization, processing, and dispatch. This is almost possible today and potentially so on current consoles, but existing APIs are a roadblock.

1.7.4 General Observations

In *Conviction*, although arbitrary occlusion tests could be issued by the main thread (to accelerate other systems, in much the same way as in *Battlefield*), we had to restrict their use in the end due to the need for deterministic behavior during co-operative play. This was primarily an issue for PC as we could not ensure matching results between GPUs from different IHVs, or indeed across generations from the same vendor. For the next title, we hope to find other applications for exploiting our system so this will not be a problem.

Were we to generate a min/max depth hierarchy, we could also return more information about the state of occlusion, which may open up more applications. By testing the z-range of objects, we can determine one or more states: Completely visible or occluded (all tests pass conclusively), partially occluded (tests pass conclusively as fully visible or occluded), potentially occluded (some tests are inclusive, i.e., z-range overlap with the HZB).

1.8 Conclusion

Whatever the future, experimenting with solutions like these is a good investment; in our experience, we gained significantly from employing these fast yet straightforward visibility systems, both in development and production terms.

The GPU implementation in particular is trivial to add (demonstrated by the fact that our initial version was developed and integrated in a matter of days) and comes with a very reasonable overhead.

1.9 Acknowledgments

We would like to thank Don Williamson, Steven Tovey, Nick Darnell, Christian Desautels, and Brian Karis, for their insightful feedback and correspondence, as well as the authors of all cited papers and presentations, for considerable inspiration.

Bibliography

[Abrash 96] Michael Abrash. "Inside Quake: Visible-Surface Determination." *Dr. Dobb's Sourcebook* Jan/Feb (1996), 41–45.

[Abrash 09] Michael Abrash. "Rasterization on Larrabee." In *Game Deveoper's Conference*, 2009.

[Akenine-Möller et al. 08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*, Third edition. Natick, MA: A K Peters, 2008.

[Andersson 10] Johan Andersson. "Parallel Futures of a Game Engine v2.0." In *STHLM Game Developer Forum*, 2010.

[Bittner et al. 09] Jiří Bittner, Oliver Mattausch, and Michael Wimmer. "Game Engine Friendly Occlusion Culling." In *ShaderX*^{7}, pp. 637–653. Hingham, MA: Charles River Media, 2009.

[Blinn 96] Jim Blinn. "Calculating Screen Coverage." *IEEE CG&A* 16:3 (1996), 84–88.

[Décoret 05] Xavier Décoret. "N-Buffers for Efficient Depth Map Query." *Computer Graphics Forum (Eurographics)* 24:3 (2005), 8 pp.

[Diamand 10] Ben Diamand. "Shadows in *God of War III*." In *Game Developer's Conference*, 2010.

[Eisemann and Décoret 06] Elmar Eisemann and Xavier Décoret. "Fast Scene Voxelization and Applications." In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, *I3D '06*, pp. 71–78. New York: ACM, 2006.

[Greene et al. 93] Ned Greene, Michael Kass, and Gavin Miller. "Hierarchical Z-buffer Visibility." In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, *SIGGRAPH '93*, pp. 231–238. New York: ACM, 1993.

[Hastings 07] Al Hastings. "Occlusion Systems." http://www.insomniacgames.com/research_dev/articles/2007/1500779, 2007.

[Hill 10] Stephen Hill. "Rendering with Conviction." In *Game Developer's Conference*, 2010.

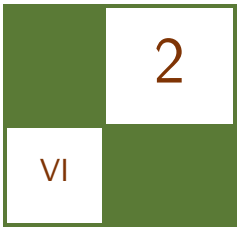
[Lloyd et al. 04] Brandon Lloyd, Jeremy Wendt, Naga Govindaraju, and Dinesh Manocha. "CC Shadow Volumes." In *ACM SIGGRAPH 2004 Sketches*, *SIGGRAPH '04*, p. 146. New York: ACM, 2004.

[Redon et al. 02] Stephane Redon, Abderrahmane Kheddar, and Sabine Coquillart. "Fast Continuous Collision Detection between Rigid Bodies." *Computer Graphics Forum* 21:3 (2002), 279–288.

[Shopf et al. 08] Jeremy Shopf, Joshua Barczak, Christopher Oat, and Natalya Tatarchuk. In *ACM SIGGRAPH 2008 Classes*, *SIGGRAPH '08*, pp. 52–101. New York: ACM, 2008.

[Soininen 08] Teppo Soininen. "Visibility Optimization for Games." Gamefest, 2008. Microsoft Download Center, Available at http://www.microsoft.com/downloads/en/details.aspx?FamilyId=B9B33C7D-5CFE-4893-A877-5F0880322AA0&displaylang=en, 2008.

[Woodard 07] Bruce Woodard. "SPU Occlusion Culling." In *SCEA PS3 Graphics Seminar*, 2007.



Shader Amortization using Pixel Quad Message Passing

Eric Penner

2.1 Introduction

Algorithmic optimization and level of detail are very pervasive topics in real-time rendering. With each rendering problem comes the question of the acceptable amount of approximation error and the quality vs. performance trade-off of increasing or decreasing approximation error. Programmable hardware pipelines play one of the largest roles in how we optimize rendering algorithms because they dictate where we can add algorithmic modification via programmable shaders.

In this article we analyze one particular aspect of modern programmable hardware—the pixel derivative instructions and pixel quad rasterization—and we identify a new level at which optimizations can be performed. Our work demonstrates how values calculated in one pixel can be passed to neighboring pixels in the frame buffer allowing us to amortize the cost of expensive shading operations. By amortizing costs in this manner we can reduce texture fetches and/or arithmetic operations by factors of two to sixteen times. Examples in this article include 4×4 percentage closer filtering (PCF) using only one texture fetch, and 2×2 bilateral upsampling using only one or two texture fetches. Our approach works using a technique we call pixel quad amortization (PQA). Although our approach already works on a large set of existing hardware, we propose some standards and extensions for future hardware pipelines, or software pipelines, to make it ubiquitous and more efficient.

2.2 Background and Related Work

As the performance of programmable graphics hardware increases exponentially, there has been a steady increase in the complexity of real-time rendering applications, often expressed as the number of arithmetic operations and texture accesses

required to shade each pixel. In response to increasing complexity, much recent research and development effort has focused on methods to reduce pixel processing workload. This includes techniques for simplifying shaders [Olano et al. 03], reusing data from previous frames [Zhu et al. 05, Nehab et al. 07] , or by simply using lower resolutions within a single frame.

One of the first upsampling approaches, known as dynamic video resizing [Montrym et al. 97], dynamically adjusts resolution based on performance, followed by simple bilinear or nearest-neighbor upsampling to a full-resolution frame. While this is effective for controlling pixel workload, artifacts are very noticeable in the upsampled frame. More recent techniques apply geometry-aware upsampling such as the joint bilateral filter [Tomasi and Manduchi 98] from either fixed size [Ren et al. 06] or dynamically resized [Yang et al. 08] frame buffers. What all of these techniques have in common is the requirement of an extra low-resolution pass, followed by upsampling. Our approach differs in that we are able to perform operations at two separate resolutions natively, in the same pass on existing hardware.

2.3 Pixel Derivatives and Pixel Quads

Before describing our technique, it is important to understand a few details of how modern graphics hardware works with respect to texture mapping, and why the pixel shader partial-derivative instructions exist. The need for partial derivatives arises from the simple problem of texture mapping a triangle. As a triangle becomes smaller on screen, one screen pixel will cover many texels, resulting in harsh aliasing unless the texture is adequately sampled. This issue is typically solved in graphics hardware with mipmapping, but a method is needed to compute which mipmap level to use.

Partial derivatives relate the infinitesimal change in one variable to the infinitesimal change in another variable at a particular location. Pixel shader partial derivatives refer to the rate of change of a shader value with respect to the screen-space x- and y-axes. When applied to texture coordinates, this can tell us how fast a texture coordinate is changing on the screen, and thus what mipmap level we should use. Before dependent texture fetches, derivatives could potentially be computed analytically, based on homogeneous barycentric texture coordinates calculated from three triangle vertices. However, dependent texture fetches can depend on arbitrary calculations including data from another texture; thus, no analytic solution exists for these cases.

The only solution remaining is to compute pixel shader derivatives discretely by looking at the value of a texture coordinate in neighboring screen pixels and computing the difference between them. Computing derivatives in this manner is called *forward differencing* or *backward differencing*, depending on whether you look at the pixel in front of or behind the current pixel, to compute the

derivative. For example, in a row of pixels p, $p[i+1]-p[i]$ is a forward difference while $p[i]-p[i-1]$ is a backward difference. Both of these typical schemes fail for parallel graphics hardware however, as they imply a dependency on the order in which pixels are computed. To solve this issue, modern hardware rasterizes triangles in quads, or 2×2 blocks of pixels, and uses custom derivative calculations that depend only on the values within a quad.

Unfortunately, neither the location of quads, nor derivatives within quads, nor even the use of pixel quads, is standardized by modern graphics APIs. Instead these details are left up to the vendor to implement as long as some form of derivative is provided. Since no documentation was provided, we turned to experimentation to determine exactly how derivatives are calculated on modern hardware. Not surprisingly, the implementations we found were exactly what one would expect, given the constraints. First and foremost, on all the hardware we tested, pixel quads have always been stationary in the same locations within the

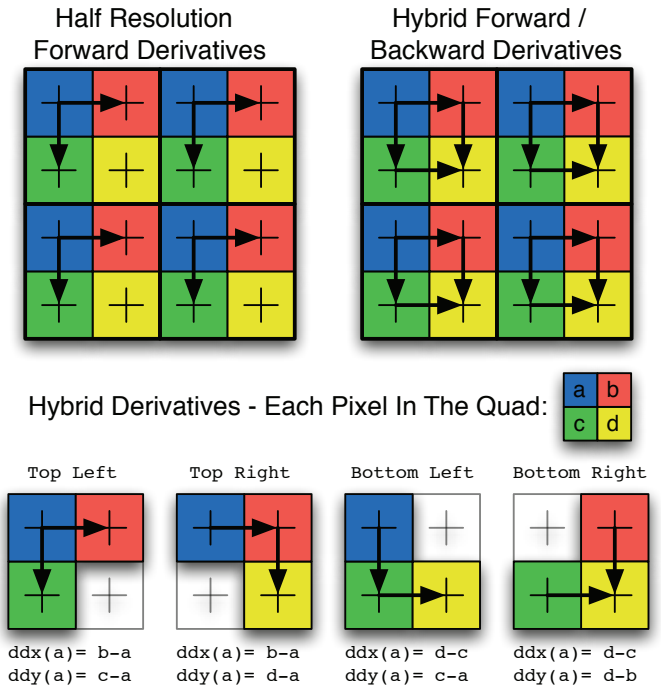


Figure 2.1. Derivative calculations used in practice in modern hardware. We found derivatives were either calculated at half-resolution using forward differencing, or using a hybrid of forward and backward differencing. In the hybrid case we have broken out each pixel's derivatives.

frame buffer; they are essentially double width pixels. Second, we found only two approaches to computing derivatives within these quads, as illustrated in Figure 2.1.

Interestingly, Shader Model 5 in Direct3D 11 has both *coarse* and *fine* versions of the derivative instructions, likely exposing the trade-offs of these two approaches to the developer. Half-resolution derivatives always return the same value within a quad, allowing for optimized texture sampling in some cases, while hybrid derivatives have the potential to provide slightly more accurate results.

It is important to note at this point that although derivative instructions were created to assist with texture mapping, they are not reserved for computing derivatives of texture coordinates. You can use the derivative instructions to calculate the derivative of *any value* in a shader. One obvious question that arises is, what happens when a triangle does not cover all the pixels in a given quad, or if some pixels in a quad are rejected by the depth test? Another question is, how does graphics hardware synchronize all the seemingly independent shader programs such that derivatives can be calculated anywhere? The answer is that in the real shader processing core the "loop" over all the pixels in a triangle is unrolled into blocks of at least four pixels. So all quad pixels are always calculated in lockstep and in parallel, likely even sharing the same set of real hardware registers. The shader program will execute for all the pixels in a quad even if only one pixel is actually needed. In the event that a quad pixel falls outside of a triangle, the values passed down from the vertices are extrapolated using the triangle's homogenous coordinates.

2.4 Pixel Quad Message Passing

Now that we have described pixel quads and why they exist in modern hardware, we turn to how we use them to our advantage. The obvious question is: Can some shaders, or some calculations within shaders, execute at the pixel quad level instead of the pixel level? If a graphics API were to theoretically support a "quad shader" it would lead to another dilemma for parallelism; we would essentially need another pipeline stage. For example, if the hardware was optimized for running 16 shaders in parallel, it would need to cache the output of 16 pixel quad shaders as input to 64 pixels shaders.

What might be a better compromise, and what we found we can already do with today's hardware, is to share values between the pixels in a given quad, but still execute a shader at the pixel level. If we choose problems that have inherent symmetries and are divisible into four identical operations, we can actually use the same pixel shader instructions to perform different "jobs" in each pixel, and then share the values in the quad. Now it should be clear why the derivative instructions are so important. They rely on the difference between two pixels, and thus can be used as a mechanism to share values between pixels, with

some simple arithmetic. We utilize the derivative instructions as message-passing instructions.

2.5 PQA Initialization

In the case of hybrid forward- and backward-differencing, each derivative is simply the positive or negative difference of the current pixel's value with the vertical/horizontal adjacent pixel. With this knowledge, it is easy to calculate what an adjacent pixel's value is. We simply subtract or add the derivative to the current pixel's value, based on the pixel's location in the quad. As an example, for the pixel in the quad in Figure 2.1(top left), we have

$$a + ddx(a) = a + (b - a) = b.$$

For the top-right pixel, we have

$$b - ddx(b) = b - (b - a) = a.$$

So to generically pass a value v horizontally within a quad and get the horizontal neighbor h, we compute

$$h = v - \text{sign}_x * ddx(v),$$

where sign_x denotes the sign of x in the quadrant of the current pixel within a quad. Although we can not access the pixel diagonally across from the current pixel directly, we can determine the horizontal neighbor followed by the vertical neighbor of that value. An example that computes all three neighbors is as follows:

```
//Gather four float4s
void QuadGather2x2(float4 value,
                   out float4 horz,
                   out float4 vert,
                   out float4 diag)
{
    horz = value + ddx(value) * QuadVector.z; //Horizontal
    vert = value + ddy(value) * QuadVector.w; //Vertical
    diag = vert  + ddx(vert)  * QuadVector.z; //Diagonal
}
```

If we need to gather only one or two values instead of a full `float4` vector, we can optimize this calculation down to as little as two MAD instructions and two derivative instructions:

```
//Gather four floats into one float4
float QuadGather2x2 (float value)
{
    float4 r = value;
    r.y  = r.x  + ddx(r.x)  * QuadVector.z; //Horizontal
    r.zw = r.xy + ddy(r.xy) * QuadVector.w; //Vertical /
        Diagonal
    return r;
}
```

In both of these examples we used the variable `QuadVector`. Figure 2.2 illustrates the value of `QuadVector` for each pixel in a quad. Most of the optimizations we perform in this chapter rely on this vector and one other variable called `QuadSelect`. `QuadVector` is used to divide two-dimensional symmetric problems into four parts, while `QuadSelect` is used to choose between two values based on the current pixel's quadrant.

The following code demonstrates one way to calculate `QuadVector` and `QuadSelect` from a pixel's screen coordinates. The negated/flipped values are also useful and are stored in z/w components.

```
void InitQuad(float2 screenCoord)
{
  //This assumes screenCoord contains an integer pixel
      coordinate
  ScreenCoord = screenCoord;
  QuadVector = frac(screenCoord.xy*0.5).xyxy;
  QuadVector = QuadVector*float4(4,4,-4,-4) + float4(-1,-1,1,1)
      ;
  QuadSelect = saturate(QuadVector);
}
```

While it takes a few instructions to initialize communication within a quad, this will allow us to amortize the cost of several costly shading operations. First, however, we will identify a few drawbacks and limitations when using PQA.

2.6 Limitations of PQA

There are a number of limitations to pixel quad amortization that become immediately apparent. First and foremost, pixel quad message passing works only on hardware that uses hybrid forward and backward derivatives as illustrated in Figure 2.2. When half-resolution derivatives are used, the derivative instructions never touch the bottom-right pixel in the quad. There is no way to communicate that pixel's value to the other pixels in the quad in that case, thus hybrid derivative support needs to be detected based on the graphics card. Appendix A

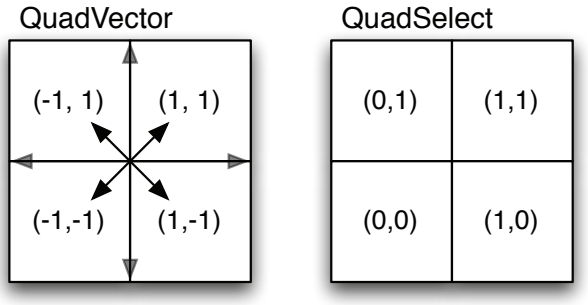


Figure 2.2. To initialize PQA we calculate two simple values for each pixel. `QuadVector` contains the x/y sign of the pixel within it's quad and is used to perform symmetric operations while `QuadSelect` is used to choose between values based on the pixel's location in the quad.

provides a list of hardware that supports hybrid derivatives at the time this article was written. It is also possible that a hardware vendor could change the way the derivative instructions work, breaking this functionality. Although this seems very unlikely, it is easy enough to write a detection routine to test which type of derivatives are used.

The second problem that becomes immediately apparent is that there is no interpolation between quads as there would be from a pre-rendered half-resolution buffer. Thus, if we output the same value for an entire quad, it will resemble unfiltered point sampling from a half-resolution frame buffer. This may be acceptable in certain situations, but if we want higher quality results, we still need to compute unique values for each pixel. Our ability to produce pleasing results really depends on the specific problem.

The third problem is that quad-level calculations work effectively only in the current triangle's domain. For example, we can use pixel quad amortization to accelerate PCF shadow-map sampling in forward rendering, but not nearly as easily in deferred rendering. This is because in the deferred case the quads being rendered are not in object space; thus, a pixel quad may straddle a depth discontinuity, creating a large gap in shadow space. In forward rendering, the entire quad will project into a contiguous location in shadow space, which is what we rely on to amortize costs effectively.

Although there are a number of drawbacks to PQA, we found we could solve these issues for several common graphics problems and still achieve large performance gains. In the following sections we will discuss how to optimize PCF, bilateral upsampling, and basic convolution and blurring with PQA.

2.7 Cross Bilateral Sampling

The cross bilateral filter has been popularized as a means to provide geometry-aware upsampling. If a screen-space buffer is blurred or upsampled using a simple bilinear filter, the features in the low-resolution buffer will bleed across depth boundaries, creating artifacts. The basic idea behind the bilateral filter is to modify the reconstruction kernel to avoid integrating across depth or normal boundaries in the scene. This is achieved by storing a depth and/or normal for each low-resolution sample and assigning filter weight according to not only the distance in screen space to each sample, but also distance in depth and/or normal space. Bilateral filters usually use Gaussian weighting functions in both depth and screen space, however [Yang et al. 08] proposed to use a simple tent function in screen space, mimicking the effect of a bilinear upsample and therefore requiring only four depth/image samples. No matter what type of weighting function is used, the filter weight is accumulated such that the sample can be normalized by the total accumulated weight:

$$c_i^H = \frac{\Sigma c_j^L f(\hat{x}_i, x_j) g(|z_i^H - z_j^L|)}{\Sigma f(\hat{x}_i, x_j) g(|z_i^H - z_j^L|)}$$

In this example $f()$ is the normal linear filtering weight while $g()$ is a Gaussian falloff based on the difference in depth between the high-resolution and low-resolution depths. One disadvantage of bilateral upsampling is its cost compared with simple bilinear filtering. While a bilinear upsample requires only one hardware filtered sample, a bilateral upsample will require at minimum four point samples and four depth samples. This cost is incurred at the high resolution, thus it often partially defeats the purpose of performing calculations at a lower resolution in the first place. Obviously, if the calculation costs less than eight samples, it will be less expensive to just compute the value at the high resolution.

The bilateral filter is one example where PQA works without any of the drawbacks mentioned in the previous section. Since bilateral upsampling occurs in screen space, we can set up our low-resolution buffer such that all the pixels in the same high-resolution quad will share the same low-resolution samples. All that is needed then is to share the samples across the quad and let each pixel perform the bilateral filter independently. Here is an example for a 2X upsample of a low-resolution AO texture. To optimize this further to only one sample, the depth can be packed into extra channels of the AO texture.

```
//Gather quad horizontal / vertical / diagonal samples
float2 AO_D, AO_D_H, AO_D_V, AO_D_D;
AO_D.x = tex2D( lowResDepthSampler , coord ).x;
AO_D.y = tex2D( lowResAOSampler    , coord ).x;
QuadGather2x2( AO_D, AO_D_H, AO_D_V, AO_D_D );
```

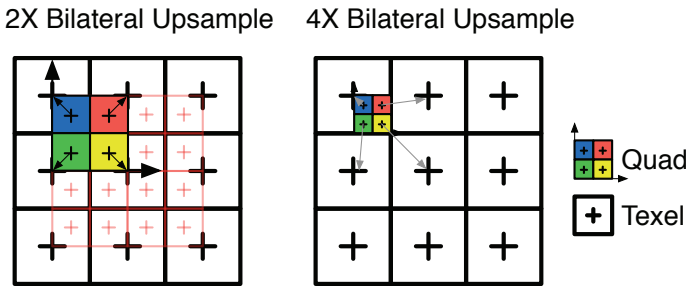


Figure 2.3. Bilateral upsampling from a half-resolution or quarter-resolution buffer. All quad pixels utilize the same four low-resolution samples. We can therefore perform a bilateral upsample with only one or two texture fetches and two derivative instructions, instead of eight texture fetches.

The bilateral upsample can then be performed as usual for each pixel, with the caveat that tent weights will need to flip to compensate for the samples being flipped in each pixel. A similar approach can be taken for a 4X upsample, or for bilateral blurring operations at any resolution. One extra thing to note is that the low-resolution buffer is shifted half a pixel (see Figure 2.3).

2.8 Convolution and Blurring

Convolution and blurring operations can also be accelerated using PQA. Although we are performing calculations at the pixel quad level, we would not want our result to be output at half-resolution or we might as well simply output a truly half-resolution texture! Thankfully, because we can share results at any point in the shader, we can customize the message delivered to other pixels in the quad in order to perform unique blurs for each pixel. The following code illustrates a 3×3 blur with four samples, while Figure 2.4 illustrates this process for a 5×5 blur using nine samples:

```
//Populate messages for neighbors
float4 m = 0;
m.rgba+= tex2D(imageSampler, coord ).x;
m.rb  += tex2D(imageSampler,coord+QuadVector*
            float2(TEXEL_SIZE.x,0)).x;
m.rg  += tex2D(imageSampler,coord+QuadVector*
            float2(TEXEL_SIZE.y,0)).x;
m.r   += tex2D(imageSampler,coord+QuadVector*
            float2(TEXEL_SIZE.xy )).x;
```

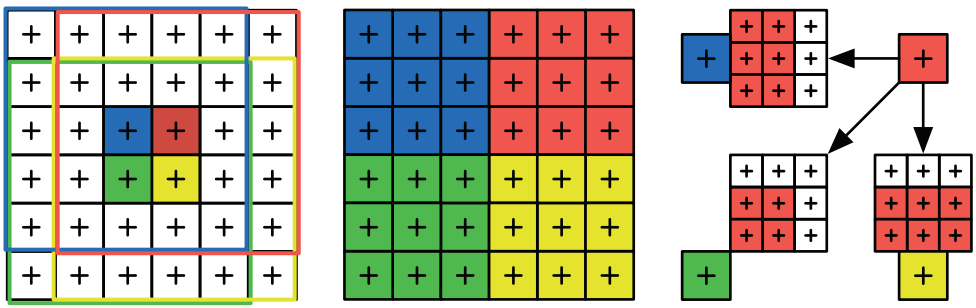


Figure 2.4. Illustration of a 5×5 blur using PQA. The blur kernel footprint of four pixels in a quad (left). Samples taken by each pixel in the quad (middle). Uniquely weighted messages from the red pixel to other pixels in the quad (right).

```
//Gather messages
float4 h, v, d;
QuadGather2x2( m, h, v, d );

//Weight results for 3x3 blur
float4 result = dot(float4(4,2,2,1) / 9.0 ,
                    float4(m.x,h.g,v.b,d.w) );
```

Unfortunately, though we can gather more samples, it becomes cumbersome to apply unique weights for more complicated filters, especially when bilinear filtering is also applied to increase the kernel width. In our example it would also take several `QuadGather` operations for a multiple channel texture. While this can be optimized significantly by separating vertical and horizontal messages, we recommend this approach primarily for performing nonseparable and/or nonlinear blurring operations on one or two channel data. In the case that only approximate results are required, we discuss a gradient approximation to support bilinear filtering in Section 2.9.

In the case of Direct3D 11 hardware, it should be noted that PQA should not be used for simple image blurring. In this case DirectCompute or OpenCL can achieve much better performance by applying the same idea in a compute shader. For example, one could output in quad-sized groups of pixels, or even output an entire row of quads in one shader. For this reason PQA should be used only during geometry rasterization on hardware that supports compute shaders. PQA will remain a valid technique in these cases since rasterization is only a semi-parallelizable task.

2.9 Percentage Closer Filtering

Percentage closer filtering refers to filtering in which a nonlinear operation is required before the filter can take place. In graphics, PCF usually refers to shadow-map filtering, where the nonlinear operation is a depth comparison. A naive N × N PCF filter looks something like this:

```
for( int i = 0; i < N; i++ )
for( int j = 0; j < N; j++ )
{
    shad += ShadowSample(Map, Coord,
                SM_TEXEL * float2( i-(N/2.0-0.5),j-(N/2.0-0.5)
                    ) );
}
shad /= (N*N);
```

Many graphics cards support native bilinear PCF filtering, and this section assumes we have at least bilinear PCF support. Some more recent graphics cards support fetching four depth values at once, allowing the user to arbitrarily filter them in the shader. Since utilizing bilinear PCF is more difficult in our case, but is supported on a much wider set of hardware, we will focus on using bilinear PCF. Extensions to `Gather` instructions can further improve results.

Since we cannot access the result of each pixel when using bilinear PCF, we start by applying an approach from [Sigg and Hadwiger 05, Gruen 10], which uses bilinear samples to build efficient larger filters. This involves using sample offsets such that each bilinear sample fetches four uniquely weighted samples. In the most simple case, where we want equal weights, this simply means placing a bilinear PCF sample in the middle of the four texels we want:

```
//Fraction of a pixel
float2 a = frac(Coord.xy * SM_SIZE - 0.5 );

//Negative/Positive offsets to compute equal weights
float4 Offset = a.xyxy * -(SM_TEXEL) +
                    float4(-0.5,-0.5,1.5,1.5)*SM_TEXEL;
float4 taps;
     taps.x = ShadowSample( Map, Coord, Offset.xw );
     taps.y = ShadowSample( Map, Coord, Offset.zw );
     taps.z = ShadowSample( Map, Coord, Offset.xy );
     taps.w = ShadowSample( Map, Coord, Offset.zy );

float shadow = dot(taps,0.25);
```

This approach can apply to arbitrary separable filters as we will see later, but for now we will keep things simple. To apply PQA, we replace the offset calculation with one that uses the quadrant vector, and then take one sample at each pixel, followed by a quad average:

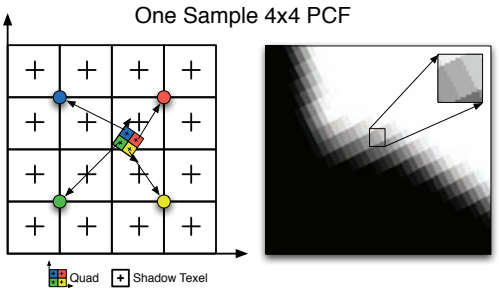


Figure 2.5. Half-resolution 4×4 PCF using quad LOD. The colored pixels correspond to the projection of one pixel quad into shadow space. Each pixel performs only one texture fetch, followed by a pixel quad average. The close-up illustrates half-resolution point sampling artifacts.

```
//Average coordinate for quad
Coord.xy = QuadAve2x2(Coord.xy);

//Fraction of a pixel
float2 a = frac(Coord.xy * SM_SIZE - 0.5 );

//Negative or positive offset to compute equal weights
float2 Offset = (-a + 0.5 + QuadVector.xy) * SM_TEXEL;
float tap = ShadowSample( Map, Coord, Offset );
float shadow = QuadAve2x2(tap);
```

We first compute the average texture coordinate for the quad. We then use the quadrant vector to select only the offset we need. Last, we take one sample in each pixel and then average the results. This is illustrated in Figure 2.5. Note that the offset calculation was also reduced from a `float4` to `float2` calculation.

At this point we are doing a lot of extra work to save only three samples, but once we extend this to larger kernels it starts to become quite effective. For example, if we use four samples per pixel we can now achieve 8×8 PCF (64 total texels) with only four bilinear samples, for a 16X improvement over the naive approach. The layout of these samples is illustrated in Figure 2.6 (right).

```
//Low and high offsets for this pixel
float4 lOhO = (-a.xyxy + QuadVector.xyxy + 0.5 +
                         float4(-2,-2,2,2) ) * SM_TEXEL;

float4 t;
       t.x = ShadowSample( Map, Coord, lOhO.xy );
       t.y = ShadowSample( Map, Coord, lOhO.xw );
       t.z = ShadowSample( Map, Coord, lOhO.zy );
       t.w = ShadowSample( Map, Coord, lOhO.zw );

float shadow = PixelAve2x2(dot(t,0.25));
```

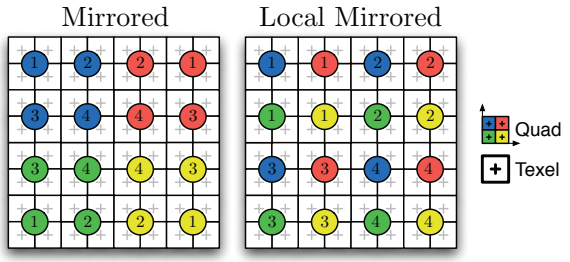


Figure 2.6. Different sample placements color coded by quad pixel. Simply mirroring samples is easier but results in samples that are very far apart, which can degrade cache performance (left). Local mirroring results in samples that are closer together but offsets can be more difficult to calculate symmetrically (right).

Although we can now sample very large kernels, we are outputting the same value for each pixel in the quad, resulting in quad-sized point sampling artifacts. Noncontinuous PCF is also quite undesirable, so it is important to add at least first-order continuity to our filter. We will now tackle both of these issues.

Higher-order filtering is more complicated since shadow texels are not located at fixed distances from the sampling location, thus weights need to be calculated dynamically. The most recent approach [Gruen 10] to achieving higher-order PCF filtering involves solving a small linear system for each sample to find the correct weights and offsets. The linear system is based on all the bilinear samples that would have touched the same texels.

We note that this can be largely simplified by using the work from [Sigg and Hadwiger 05]. Instead of replicating the weights produced by several bilinear samples and a grid of weights, we determine the weight for each texel using an analytic filter kernel. Because the kernel is separable, we can compute the sample offsets and weights separately for each axis. This is demonstrated using a full-sampled Gaussian kernel below.

```
{
    #define SIGMA (SM_TEXEL*2)
    #define ONE_OVER_TWO_SIGMA_SQ (1.0/(2.0*SIGMA*SIGMA))
    #define GAUSSIAN(v) (exp(-(v*v)*ONE_OVER_TWO_SIGMA_SQ))

    float4 GaussianFilterWeight(float4 offset)
    {
    return GAUSSIAN(offset) - GAUSSIAN(4*SM_TEXEL);
    }

    float4 linstep(float4 min, float4 max, float4 v)
    {
    return saturate((v-min)/(max-min));
    }
```

```
float4 LinearStepFilterWeight(float4 offset)
{
return linstep(SM_TEXEL*4.0,SM_TEXEL*2.0,abs(offset));
}

float4 SmoothStepFilterWeight(float4 offset)
{
return smoothstep (SM_TEXEL*4.0,SM_TEXEL*1.0,abs(offset));
}

float4 FilterWeight(float4 offset, const int filterType)
{
switch(filterType)
    {
    case 0:
    return LinearStepFilterWeight(offset,texelWidth);
    case 1:
    return SmoothStepFilterWeight(offset,texelWidth);
    case 2:
    return GaussianFilterWeight(offset,texelWidth);
    }
}

float Shadow8x8Hlaf(Texture2D Map, float4 Coord, const int
    filterType)
{
    //Compute average coord, and fraction of pixel
    Coord = QuadAve(Coord);
    float2 a = frac(Coord.xy*SM_SIZE - 0.5);

    //Low and high pixel center offsets (local mirrored)
    float4 offsets0 = (-a.xyxy + QuadVector.xyxy + float4
        (-2,-2,2,2))*SM_TEXEL;
    float4 offsets1 = offsets0 + SM_TEXEL;

    //Filter weights and offsets
    float4 g0 = FilterWeight(offsets0,filterType);
    float4 g1 = FilterWeight(offsets1,filterType);
    float4 g01 = g0 = g1;
    float4 bilinearOffsets = offsets0 + (g1/g01)*SM_TEXEL;

    //Gather 64 shadow map texels with 4 samples
    float4 taps;
    taps.x = ShadowSample(Map, Coord, bilinearOffsets.xy);
    taps.y = ShadowSample(Map, Coord, bilinearOffsets.zy);
    taps.z = ShadowSample(Map, Coord, bilinearOffsets.xw);
    taps.w = ShadowSample(Map, Coord, bilinearOffsets.zw);
    float4 weights = g01.xzxz*g01.yyww;

    //Sum weights and samples across the quad.
    float4 shadow_weight;
    shadow_weight.x = dot(taps,weights);
    shadow_weight.y = dot(1,weights);
```

```
            shadow_weight.xy = QuadAve(shadow_weight.xy);

            //Normalize our sample weight
            float shadow = shadow_weight.x/shadow_weight.y;
            return shadow;
        }
    }
```

We have shown a few Gaussian filters for both simplicity and readability; in practice, we prefer to use linear, quadratic, or cubic B-spline kernels. Note that we do not need to calculate weights for each texel, but rather for each row and column of texels. Bilinear offsets can then similarly be computed separately and weights simplified to the product between the sum of X and Y weights. The same approach can be applied for piecewise polynomial filters such as B-Splines, or using arbitrary filters with the offsets and weights stored in lookup textures as in [Sigg and Hadwiger 05].

At this point we now have very smooth shadows but still have the same value for all pixels in a quad. To smooth the point-sampled look, it would be optimal to bound all quad texels in shadow space and create a uniquely weighted kernel for each pixel, but without `Gather()` capability that would involve performing four

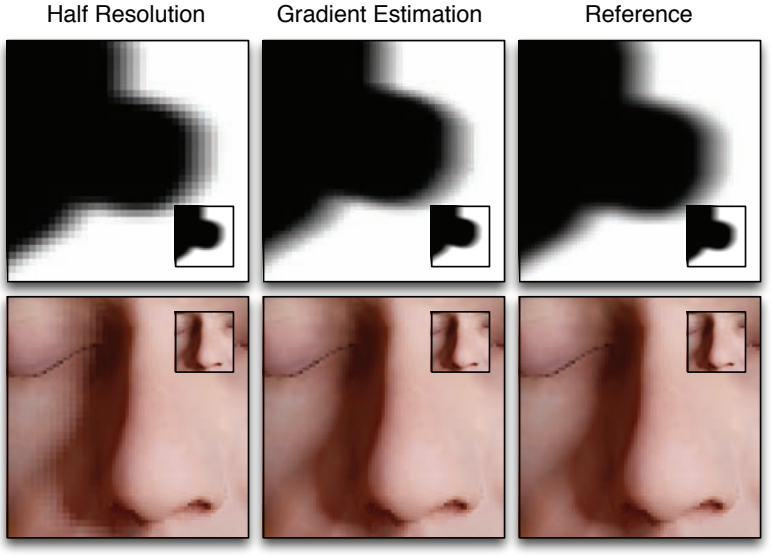


Figure 2.7. Gradient estimation for 8×8 bilinear PCF (four samples). These images are magnified to illustrate how even very naive gradient estimation can hide most quad artifacts. If using Shader Model 4 or 5, `Gather` samples can be used to avoid these artifacts altogether.

times the samples. We found that a good compromise when using bilinear PCF is to compute a simple gradient approximation along with the shadow value. Rather than using every texel to compute the gradient, we simply reuse the bilinear-filtered samples as if they had come from a lower-resolution shadow map. Although this is somewhat of a hack, thankfully it actually works quite well (see Figure 2.7). The weights for the derivative calculation will depend on the kernel itself (see Figure 2.8). The following code calculates a 4×4 Prewitt gradient which works well for low-order B-spline filters:

```
//Gradient estimation using Prewitt 4x4 gradient operator
float4 s_dxdy;
s_dxdy.xy = dot( taps, weights );
//Prewitt (x)
s_dxdy.z  = dot( taps, float4(3,3,1,1) * QuadVector.x );
//Prewitt (y
s_dxdy.w  = dot( taps, float4(3,1,3,1) * QuadVector.y );)
s_dxdy = QuadAve( s_dxdy ) * 4;
float shadow = s_dxdy.x;
```

Figure 2.8. All of the images in Part II, Chapter 1 also make use of PQA for PCF shadows. This image uses 8×8 bilinear PCF filtering (four samples) and half the original ALU operations. Part of the shadow penumbra is used to mimic scattering fall-off, thus an inexpensive wide PCF kernel is crucial. Mesh and textures courtesy of XYZRGB.

```
shadow += dot((Coord.xy-realCoord), s_dxdy.zw );
return shadow;
```

The last issue that needs to be mentioned is handling anisotropy and minification. Our gradient estimate works well on close-ups and will handle minification up to the size of the kernel used. However, under extreme minification the distance between the quad pixels in shadow space increases, and the linear gradient estimate breaks down. There are a number of ways we can deal with this. Firstly, if using a technique like cascading shadow maps (CSMs) we are unlikely to experience extreme minification since shadow resolution should be distributed somewhat equally in screen space. In other cases, one option is to generate mipmaps of the shadow map, allowing us to increase the kernel size to fit the footprint of the quad in shadow space. Alternatively, we can also forgo generating mipmaps and just sparsely sample a larger footprint in the shadow map. We have found that both of these solutions work adequately. Again, having `Gather()` support opens up several more options.

2.10 Discussion

We have demonstrated a new approach for optimizing shaders, by amortizing costly operations across pixel quads, that is natively supported by a large set of existing hardware. Our approach has the advantage of not requiring additional passes over the scene unlike other frame buffer LOD approaches. It also potentially allows for sharing redundant calculations and temporary registers between pixels, while still performing the final calculation at full resolution. We have also demonstrated how gradients can be used to generate smooth results within a quad while still supporting bilinear texture fetches. The primary drawback of our approach remains the lack of interpolation between neighboring quads that would be provided with something like bilateral upsampling. Interestingly, however, our technique can help in either case, since our technique can also accelerate the bilateral upsampling operation itself.

Should PQA become a popular technique, hardware or software pipelines could make it much more efficient by exposing the registers of neighboring pixels directly in the pixel shader. Native API support for sharing registers between pixels would greatly simplify writing amortized shaders. The current cost of sharing results via derivative instructions makes it prohibitive in some cases.

We have found that our approach can also be applied to other rendering problems, such as shadow-contact hardening, ambient occlusion, and global illumination. Although we can not verify this at the time of writing, it also appears that all future hardware that supports Direct3D 11's fine derivatives will support PQA.

2.11 Appendix A: Hardware Support

At the time of writing we have verified that PQA works on all recent NVIDIA hardware. We have tested several 8000 and 9000 series cards including mobile cards in laptops. Unfortunately all the ATI hardware we have tested so far, including the Xbox 360, do not support PQA as they use half-resolution derivatives. The PlayStation 3 console, on the other hand, does support PQA since it uses an NVIDIA GPU, making PQA feasible for current console games. Since Direct3D 11 specifies two types of derivatives, PQA will likely be supported by all hardware that supports Shader Model 5. At the time of writing we do not have access to any Intel graphics cards and thus we do not know which form of derivative Intel graphics cards use.

To detect if message passing works on an arbitrary card, we draw a small `rect` with a custom shader and look at (or read back) the results. The custom shader sets a variable to four in only one quad pixel and zero otherwise. The result of calling `QuadAve()` on that variable will be one for all pixels if message passing worked and something else otherwise. This test is repeated for all quad pixels.

Bibliography

[Gruen 10] Holger Gruen. "Fast Conventional Shadow Filtering." In *GPU Pro: Advanced Rendering Techniques*, pp. 415–445. Natick, MA: A K Peters, 2010.

[Montrym et al. 97] J.S. Montrym, D.R. Baum, D.L. Dignam, and C.J. Migdal. "InfiniteReality: A Real-Time Graphics System." In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, *SIGGRAPH '97*, pp. 293–302. New York: ACM Press/Addison-Wesley Publishing Co., 1997.

[Nehab et al. 07] Diego Nehab, Pedro V. Sander, Jason Lawrence, Natalya Tatarchuk, and John R. Isidoro. "Accelerating Real-Time Shading with Reverse Reprojection Caching." In *ACM Siggraph/Eurographics Symposium on Graphics Hardware*, pp. 25–35. Aire-la-Ville, Switzerland: Eurographics Association, 2007.

[Olano et al. 03] Marc Olano, Bob Kuehne, and Maryann Simmons. "Automatic shader level of detail." In *ACM Siggraph/Eurographics Conference on Graphics Hardware*, pp. 7–14. Aire-la-Ville, Switzerland: Eurographics Association, 2003.

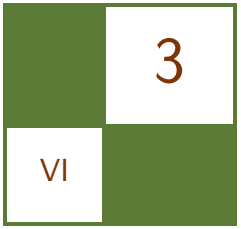
[Ren et al. 06] Zhong Ren, Rui Wang, John Snyder, Kun Zhou, Xinguo Liu, Bo Sun, Peter-Pike Sloan, Hujun Bao, Qunsheng Peng, and Baining Guo. "Real-Time Soft Shadows in Dynamic Scenes using Spherical Harmonic Exponentiation." *ACM Siggraph Transactions on Graphics* 25:3 (2006), 977–986.

[Sigg and Hadwiger 05] Christian Sigg and Markus Hadwiger. "Fast Third-Order Filtering." In *GPU Gems 2*, Chapter 20. Reading, MA: Addison-Wesley Professional, 2005.

[Tomasi and Manduchi 98] C. Tomasi and R. Manduchi. "Bilateral Filtering for Gray and Color Images." In *Proceedings of the Sixth International Conference on Computer Vision*, *ICCV '98*, pp. 839–. Washington, DC: IEEE Computer Society, 1998.

[Yang et al. 08] Lei Yang, Pedro V. Sander, and Jason Lawrence. "Geometry-Aware Framebuffer Level of Detail." 27:4 (2008), 1183–188.

[Zhu et al. 05] T. Zhu, R. Wang, and D. Luebke. "A GPU Accelerated Render Cache." In *Pacific Graphics*, 2005.



A Rendering Pipeline for Real-Time Crowds

Benjamín Hernández and Isaac Rudomin

In motion pictures, large crowds of computer-generated characters are usually included to produce battle scenes of epic proportions. In real-time strategy games, it is common to find crowds as armies controlled by users (or AI), or crowds made up of non-player characters (e.g., groups of spectators in a stadium). In virtual environments, it is common to find crowd simulations that interact with other characters and with their surrounding environment.

In all cases, optimizations such as level of detail and culling should be performed to render the crowds. In this chapter, we propose a parallel approach (implemented on the GPU) for level of detail selection and view-frustum culling, allowing us to render crowds made up of thousands of characters.

3.1 System Overview

Our rendering pipeline is outlined in Figure 3.1. First, all necessary initializations are performed on the CPU. These include loading information stored on disk (e.g., animation frames and polygonal meshes) and information generated as a preprocess (e.g., character positions) or in runtime (e.g., camera parameter updates). This information is used on the GPU to calculate the characters' new positions, do view-frustum culling, assign a specific level of detail (LOD) for each character and for level of detail sorting and character rendering. A brief description of each stage is given here.

- *Populating the virtual environment and behavior.* In these stages we specify the initial positions of all the characters, how they will move through the virtual environment, and how they will interact with each other. The result is a set of updated character positions.

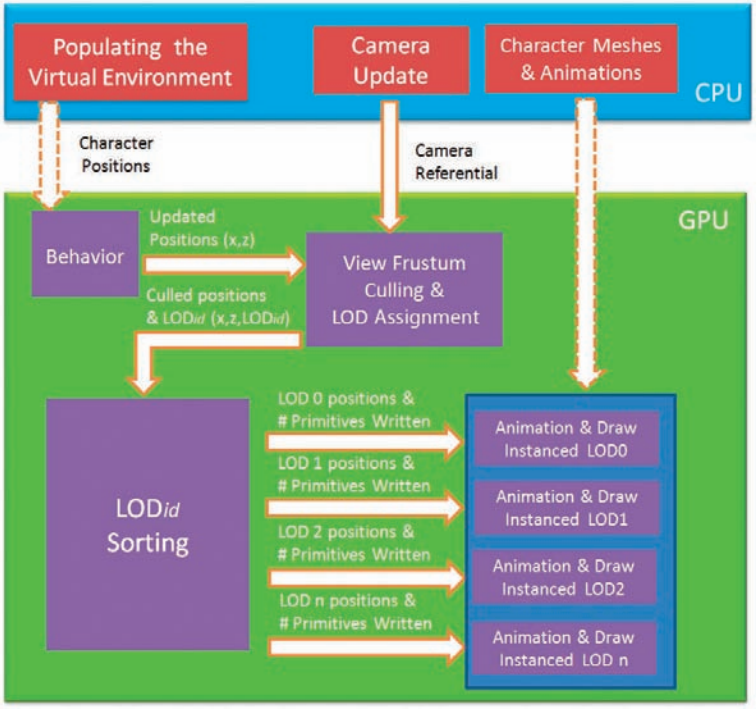


Figure 3.1. Rendering pipeline for crowd visualization. Dashed arrows correspond to data transferred from main memory to GPU memory only once at initialization.

- *View-frustum culling and LOD Assignment.* In this stage we use the characters' positions to identify those that will be culled. Additionally, we assign a proper LOD identifier to the characters' positions inside the view frustum according to their distance to the camera.
- *LOD sorting.* The output of the view-frustum culling and LOD assignment stage is a mixture of positions with different LODs. In the LOD sorting stage we sort each position, according to its LOD identifier, into appropriate buffers so that all the characters' positions in any one buffer have the same level of detail.
- *Animation and draw instancing.* In this stage we will use each sorted buffer to draw the appropriate LOD character mesh, using instancing. Instancing allows us to translate the characters across the virtual environment and add visual and geometrical variety to the individuals that form the crowd.

In the following sections, we will present a detailed description of how we implemented these stages.

3.2 Populating the Virtual Environment and Behavior

For simplicity, our virtual environment is a plane. It is parallel to the plane formed by the xz-axes. The initial positions of the characters are calculated randomly and stored into a texture.

For behavior, we implemented finite-state machines (FSMs) as fragment shaders. A FSM is used to update the characters' positions following [Rudomín et al. 05, Millán et al. 06], in which a character will consult the value of a labeled world map and follow a very simple FSM that causes it to move right until it reaches the right edge of the map, at which point the agent changes state and starts moving left until it gets to the left also ideal for this pipeline.[1]

Implementing FSM as fragment shaders needs three kinds of textures: a world-space texture, an agent texture, and an FSM-table texture. World-space textures encode values for each location in the virtual environment. This covers many types of maps: heightmaps, collision maps, interest area maps, or action maps. We consider these maps as maps labeled with some value on each pixel. Agent textures have a pixel for each character and encode the state s of the character and its position (x, z) in the world map. Finally, the finite state machine is represented as a texture where given a certain state of the character and a certain input, we can obtain a new state and position of the character following the basic algorithm shown in Listing 3.1.

```
given agent i
    state=agent[i].s; x=agent[i].x; z=agent[i].z;
    label=world[x,z];
    agent[i].s=fsm[state,label];
    agent[i].x += fsm[state,label].delta_x;
    agent[i].z += fsm[state,label].delta_z;
```

Listing 3.1. Basic algorithm to implement FSM as fragment shader.

3.3 View-Frustum Culling

View-frustum culling (VFC) consists of eliminating groups of objects outside the camera's view frustum. The common approach for VFC is to test the intersection between the objects and the six view-frustum planes using their plane equations to determine the visibility of each object. In our case, we implement a simpler

[1] The reason we recommend methods that use the GPU for behavior, in addition to the fact that these methods can simulate the behavior of tens of thousand characters efficiently, is that approaches using the GPU eliminate the overhead of transferring the new characters' positions between the CPU and and the GPU on every frame.

method called radar VFC [Puig Placeres 05]. Radar VFC is based on the camera's referential points. The method tests the objects for being in the view range or not, thus there is no need to calculate the six view-frustum plane equations.

On the other hand, objects tested against the view frustum are usually simplified using points or bounding volumes such as bounding boxes (oriented or axis-aligned) or spheres. In our case, we use points (the characters' positions) together with radar VFC to perform only three tests to determine the characters' visibility. In addition, to avoid the culling of characters that are partially inside the view frustum, we increase the view frustum size by Δ units[2] (Figure 3.2).

As mentioned earlier, radar VFC is based on camera referential points. In other words, the camera has a referential based on the three unit vectors $\hat{x}$, $\hat{y}$, and $\hat{z}$ as shown in Figure 3.3, where c is the position of the camera, n is the center of the near plane, and f is the center of the far plane.

The idea behind radar VFC is that once we have the character's position p to be tested against the view frustum, we find the coordinates of p in the referential and then use this information to find out if the point is inside or outside the view frustum.

The first step is to find the camera's referential. Let d be the camera's view direction, $\hat{u}$ the camera's up vector, then unit vectors $\hat{x}$, $\hat{y}$, and $\hat{z}$ that form the referential are calculated using Equations 3.1, 3.2, and 3.3.

$$\hat{z} = \frac{d}{\|d\|} = \frac{d}{\sqrt{d_x^2 + d_y^2 + d_z^2}} \tag{3.1}$$

$$\hat{x} = \frac{\hat{z} \otimes \hat{u}}{\|\hat{z} \otimes \hat{u}\|} \tag{3.2}$$

$$\hat{y} = \frac{\hat{x} \otimes \hat{z}}{\|\hat{x} \otimes \hat{z}\|} \tag{3.3}$$

Once we have calculated the referential, the next step is to compute the vector v that goes from the camera center c to the agent's position p using Equation 3.4:

$$v = p - c. \tag{3.4}$$

Next, the vector v is projected onto the camera referential, i.e., onto the $\hat{x}$, $\hat{y}$, and $\hat{z}$ unit vectors.

Radar VFC first tests vector v against $\hat{z}$; v is outside the view frustum if its projection $\text{proj}_{\hat{z}} v \notin (\text{nearPlane}, \text{farPlane})$. Notice that the projection of a vector a into a unit vector $\hat{b}$ is given by the dot product of both vectors, i.e., $\text{proj}_{\hat{b}} a = a \cdot \hat{b}$.

If $\text{proj}_{\hat{z}} v \in [\text{nearPlane}, \text{farPlane}]$, then vector v is tested against $\hat{y}$; v will be outside the view frustum if its projection $\text{proj}_{\hat{y}} v \notin (-(h/2+\Delta), h/2+\Delta)$ interval,

[2]The value of Δ is obtained by visually adjusting the view frustum.

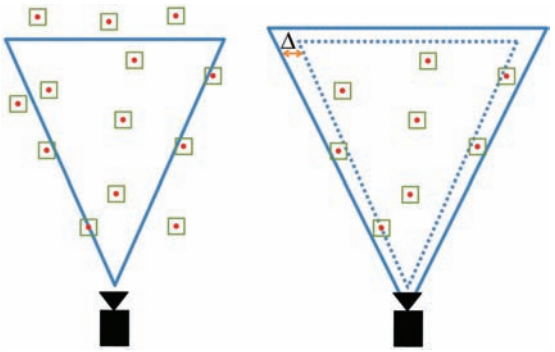


Figure 3.2. View-frustum culling.

where h is the height of the view frustum at position v and Δ is the value used to increase the view-frustum size as shown in Figure 3.2. The height h is calculated using Equation 3.5, where fov is the field-of-view angle:

$$h = \text{proj}_{\hat{z}} v \times 2 \times \tan \frac{\text{fov}}{2} : \text{fov} \in [0, 2\pi] \tag{3.5}$$

If $\text{proj}_{\hat{y}} v \in (-(h/2 + \Delta), h/2 + \Delta)$, then vector v is tested against $\hat{x}$ (i.e., v is outside the view frustum if its projection $\text{proj}_{\hat{x}} v \notin (-(w/2 + \Delta), w/2 + \Delta)$ interval) where w is the width of the view frustum, given in Equation 3.6 and ratio is the aspect ratio value of the view frustum:

$$w = h \times \text{ratio} \tag{3.6}$$

VFC and LOD assignment stages are performed using a geometry shader. This shader receives as input the agent texture that was updated in the behavior stage (Section 3.2), and it will emit the positions (x, z) which are inside the view

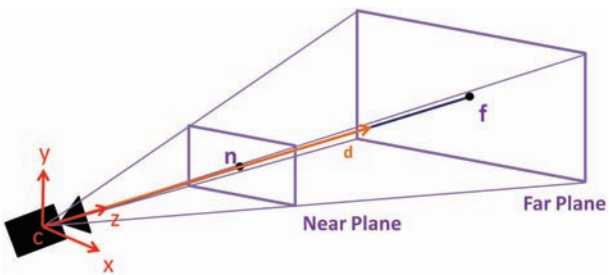


Figure 3.3. Camera's referential based on the three unit vectors x, y, and z.

frustum and a $\mathrm{LOD}_{\mathrm{id}}$. The resultant triplets $(x, y, \mathrm{LOD}_{\mathrm{id}})$ are stored in a vertex buffer object using the OpenGL transform feedback feature. Listing 3.2 shows the code that performs radar VFC in GLSL.

```
[vertex program]
void main(void)
{
   gl_TexCoord[0] = gl_MultitexCoord0;
   gl_Position = gl_Vertex;
}
[geometry program]
#define INSIDE  true
#define OUTSIDE false
uniform sampler2DRect position;
uniform float nearPlane, farPlane, tang, ratio, delta;
uniform vec3 camPos, X, Y, Z;

bool pointInFrustum(vec3 point)
{
   // calculating  v = p - c
   vec3 v = point - camPos;
   // calculating the projection of v into Z unit vector
   float pcz = dot(v,Z);


   // First test: test against Z unit vector
   if (pcz > farPlane || pcz < nearPlane)
      return OUTSIDE;

   // calculating the projection of v into Y unit vector
   float pcy = dot(v,Y);
   float h = pcz * tang;
   h = h + delta;

   // Second test: test against Y unit vector
   if (pcy > h || pcy < -h )
      return OUTSIDE;

   // calculating the projection of v into X unit vector
   float pcx = dot(v,X);
   float w = h * ratio;
   w = w + delta;

   // Third test: test against X unit vector
   if (pcx > w || pcx < - w )
      return OUTSIDE;

   return INSIDE;
}
```

```
void main(void)
{
    vec4 pos = texture2DRect(position,gl_TexCoordIn[0][0].st);
    if (pointInFrustum(pos.xyz))
    {
        gl_Position = pos;
        EmitVertex();
        EndPrimitive();
    }
}
```

Listing 3.2. Code for radar view frustum culling in GLSL.

3.3.1 Assigning Level of Detail

After determining which positions are inside the view frustum, the next step is to assign a LOD_{id} according to a given metric. In this case, we use discrete LOD[3] which consists of creating different LODs for each character as a preprocess. At runtime, the appropriate character's LOD is rendered using its LOD_{id}.

Metrics for assigning values to LOD_{id} can be based on distance to the camera, model size in screen space, eccentricity of the model with respect to the camera, or perceptual factors, among others. For performance and simplicity, we are using the distance to the camera as our metric; we also use visual perception to reduce the popping effect. The idea behind the distance to the camera metric is to select (or in our case, assign) the appropriate LOD based on the distance between the model and the viewpoint (i.e., coarser resolution for distant geometry). Nevertheless, instead of computing the Euclidean distance between the object and the viewpoint, we define the appropriate LOD as a function of the view range and the far plane. These values are obtained by the camera referential points.

The camera's view range is given by unit vector $\hat{z}$ and it is limited by the distance between the camera center, c, and the farPlane value. Thus, we test the projection of v onto $\hat{z}$ ($\text{proj}_{\hat{z}}v$), against different fixed intervals of the view range to assign a value to LOD_{id}.

A common approach for manually assigning values to LOD_{id} is using `if` statements as shown in Listing 3.3. Nevertheless, we can reduce GPU branching by eliminating the `if` statements and by using a sum of unit step functions instead (Equation 3.7):

$$\text{LOD}_{\text{id}} = \sum_{i=0}^{n-1} U(\text{proj}_{\hat{z}}v - \text{farPlane} \times \tau_i) \tag{3.7}$$

[3]It has been shown in [Millán et al. 06] that 2D representations, such as impostors, make it possible to render tens of thousands of similar animated characters, but 2D-representation approaches need manual tuning and generate a huge amount of data if several animation sequences are present and/or geometrical variety is considered.

```
...
if projZv <= range0 then
    LODid = 0
else if projZv > range0 & projZv <= range1 then
    LODid = 1
else if projZv > range1 & projZv <= range2 then
    LODid = 2
...
```

Listing 3.3. Assigning LOD_{id} using `if` statements.

where n is the number of LOD meshes per character, $\tau_i \in (0, 1)$, is a threshold that isotropically or anisotropically divides the view range visually calibrated to reduce popping effects and U is the unit step function given by:

$$U(t - t_0) = \begin{cases} 1 & \text{if } t \geq t_0, \\ 0 & \text{if } t < t_0. \end{cases}$$

Notice that if $n = 3$ (three LOD meshes per character), then LOD_{id} can receive three values, 0 when the characters are near the camera (full detail), 1 when the characters are at medium distances from the camera (medium detail) and 2 when the characters are at distances far from the camera (low detail).

Listing 3.4 shows the changes made in Listing 3.2 to add LOD_{id} calculation.

```
[geometry shader]
...
bool pointInFrustum(vec3 point, out float lod)
{
...
   // calculating the projection of v into Z unit vector
   float pcz = dot(v,Z);
...
   // For 3 LOD meshes:
   lod = step(farPlane*tao0, pcz) +
         step(farPlane*tao1, pcz) +
         step(farPlane*tao2, pcz);

   return INSIDE;
}

void main(void)
{
   float lod;
```

```
    vec4 pos = texture2DRect(position,gl_TexCoordIn[0][0].st);
    if (pointInFrustum(pos.xyz, lod))
    {
        gl_Position = pos;
        gl_Position.w = lod;
        EmitVertex();
        EndPrimitive();
    }
}
```

Listing 3.4. Assigning LOD_{id} using `step` functions.

3.4 Level of Detail Sorting

The result of the VFC and LOD assignment stage is that we have filled up a vertex buffer object (VBO) for all of the characters with positions inside the camera's view frustum. IBased on their distances to the camera, we have assigned a LOD_{id} for each position (Figure 3.4(a)). On the other hand, hardware instancing requires a single LOD mesh to draw several instances of the same mesh, thus we need to organize these positions according to their LOD_{id} in different VBOs.

Following [Park and Han 09], we will sort the output VBO from the VFC and LOD assignment stage into appropriate VBOs (that we will call $VBOs_{LOD}$) such that all of the characters' positions in a VBO have the same LOD_{id} (Figure 3.4(b)). Since we are using three LODs, we will use three VBOs.

In this case, we use transform feedback to populate each VBO_{LOD}. In total, we perform three transform feedback passes. In addition, transform feedback allows us to know how many primitives were written in each VBO_{LOD}. The number of primitives written will be used when calling the `Draw Instanced` routine.

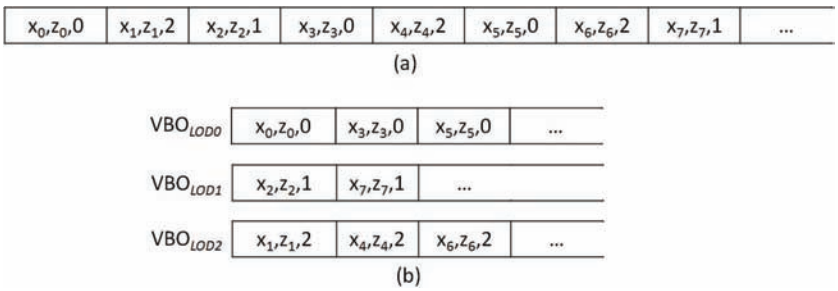


Figure 3.4. (a) Output VBO from VFC and LOD assignment stage. (b) Output of LOD sorting stage.

For each transform feedback pass, a geometry shader will emit only the vertices of the same LOD_{id}. This is shown in Listing 3.5. Notice that the uniform variable `lod` is updated each pass. In our case it will be set to 0 for a full-resolution mesh, 1 for a medium-resolution mesh, and 2 for a low-resolution mesh.

```
[geometry shader]
uniform float lod; // this variable is updated each pass
void main ()
{
    vec4 pos = gl_PositionIn[0];
    if ( lod == pos.w  )
    {
        gl_Position =  pos;
        EmitVertex();
        EndPrimitive();
    }
}
```

Listing 3.5. Geometry shader used to populate each VBO_{LOD}.

Figure 3.5 shows the output of this stage and the VFC and LOD assignment stage. The characters' positions are rendered as points. We have assigned a specific color for each VBO_{LOD}. In this case, red was assigned to VBO_{LOD0}, green to VBO_{LOD1}, and blue to VBO_{LOD2}.

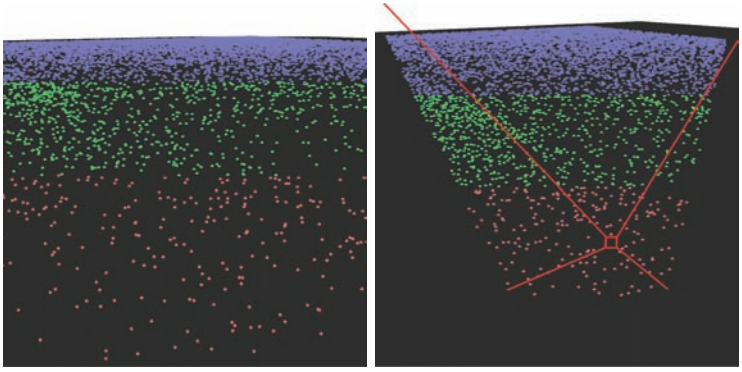


Figure 3.5. Output of LOD sorting stage, 4096 characters rendered as points. LOD0 is shown in red, LOD1 in green, and LOD2 in blue. Main camera view (left). Auxiliary camera view; notice that only positions inside the view frustum are visible (right).

3.5 Animation and Draw Instanced

As a preprocess, we load all of the character meshes and textures that will form the crowd. We also define several character groups according to their geometrical appearance. Animation is performed with a technique that reuses a character's "rig" on the other characters of the group. With this method, the animation sequence is stored in a texture array in which each layer stores an animation frame. Animation frames specify the rotation angles of the character's joints. These angles are used to pose a character, and by interpolating them, we perform character animation totally on the GPU. However, approaches such as AniTextures [Bahnassi 06] and skinned instancing [Dudash 07] can be used as an alternative.

At runtime, for each character group, we render the high-resolution instances first using the positions stored in `VBO_LOD0` to world-transform each instance; then we render the medium-resolution instances using `VBO_LOD1`, and finally the low-resolution instances using `VBO_LOD2`. In each call, we use the function `glDrawElementsInstanced` available in OpenGL. This function generates a unique instance value called `gl_InstanceId`, which is accessible in a vertex shader and t is used as an index to access the instance's specific position, animation frame, and its visual characteristics.

3.6 Results

We designed two tests to verify the performance of our pipeline. These tests were performed on Windows Vista using an NVIDIA 9800GX2 card with SLI-disabled and a viewport size of 900×900 pixels.

The goal of the first test is to determine the execution time of the behavior, VFC and LOD assignments, and LOD sorting stages.[4] The goal of the second test is to determine the execution time of the complete pipeline. The first test consisted of incrementing the number of characters from 1K to 1M, each character with three LODs. Timing information was obtained using timer queries (GL_EXT_timer_query) which provides a mechanism used to determine the amount of time (in nanoseconds) it takes to fully complete a set of OpenGL commands without stalling the rendering pipeline.

Results of this test are shown in the graph in Figure 3.6 (timing values are in milliseconds). In addition, Figure 3.5 shows a rendering snapshot for 4096 characters rendered as points. Notice that the elapsed time for VFC and LOD assignments and LOD sorting stages remains almost constant. When performing transform feedback, we do not need any subsequent pipeline stages, thus rasterization is disabled.

The second test consists of rendering a crowd of different characters. Each character has three LODs, the character's LOD0 mesh is made of 2500 vertices,

[4]We do not provide the execution time of the animation and draw instanced stage, since timing results are bigger by several orders of magnitude.

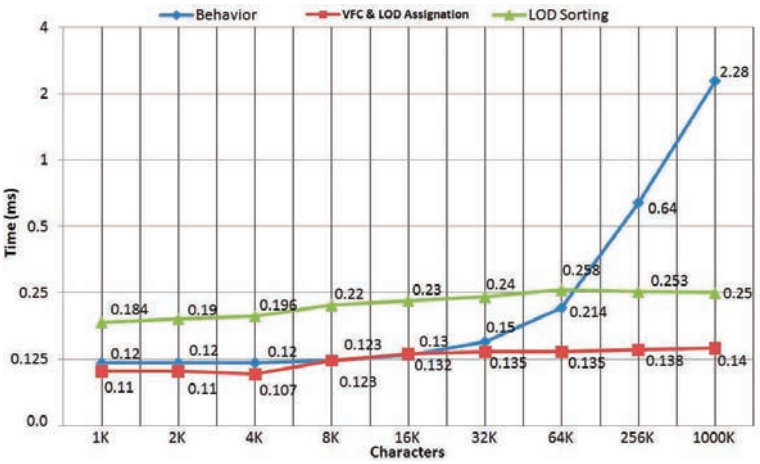


Figure 3.6. Test 1 results. Notice that timing results are in milliseconds.

the LOD1 mesh 1000 and LOD2 300. The goal of this test is to determine the execution time of all the stages of our pipeline using two different camera perspectives. In Perspective A (Figure 3.7), almost all characters are visible, while in Perspective B (Figure 3.8) almost all characters are culled.

Figure 3.7. Perspective A (8192 characters). Most of the characters are visible.

Figure 3.8. Perspective B (8192 characters). Most of the characters are culled.

Agents	t $\times 10^{-3}$	LOD0	LOD1	LOD2	Vertices $\times 10^{6}$	Visible %	Culled %
1024	47.62	274	644	54	1.35	95	5
2048	62.11	365	688	397	1.70	71	29
4096	95.60	450	1388	943	2.80	68	32
8192	159.24	483	1438	5085	4.17	86	14
12288	194.93	400	1905	7694	5.21	81	19
16384	261.78	476	2477	10388	6.78	81	19

Figure 3.9. Results obtained in Perspective A.

Agents	t $\times 10^{-3}$	LOD0	LOD1	LOD2	Vertices $\times 10^{6}$	Visible %	Culled %
1024	19.49	161	0	0	0.40	16	84
2048	22.91	204	0	0	0.51	10	90
4096	49.33	412	314	0	1.34	18	82
8192	57.77	503	317	0	1.60	10	90
12288	72.99	471	995	0	2.20	12	88
16384	97.18	541	1546	0	2.90	13	87

Figure 3.10. Results obtained in Perspective B

These results are shown in Table 3.9 for Perspective A, and in Table 3.10 for Perspective B. The first column of both tables shows the number of rendered characters, the second one shows the time, in milliseconds, measured for each case. Columns three to five show how many characters per level of detail are rendered, and column six shows the total number of vertices, in millions, transformed by our animation shader. Finally, the last two columns show the percentage of characters that are visible or culled.

3.7 Conclusions and Future Work

We have shown that optimization techniques such as view-frustum culling and LOD selection in the GPU result in a very small time penalty. In our practical case, the stage that took more time to execute was animation and draw instanced, which was to be expected. Moreover, extra memory requirements do not exceed the amount needed to store a 32-bit floating texture of 512×512 pixels (i.e., for sixteen thousand characters we needed to allocate four floating-point vertex-buffer objects of 128×128, one auxiliary vertex-buffer object to store partial results obtained from the VFC and LOD assignment stage, and three vertex buffers to store the positions of each level of detail.)

However, performance results can be improved and memory requirements can be reduced by using the new OpenGL 4.0 characteristic called "multiple transform feedback," contained in the `ARB_transform_feedback3`, `ARB_gpu_shader5` and `NV_gpu_program5` extensions, which allows geometry shaders to direct each vertex arbitrarily to a specified vertex stream. Therefore, we will require only one transform feedback call for LOD_{id} sorting, and by combining the VFC and LOD assignment and LOD sorting stages we could dispense with the auxiliary vertex-buffer object used to store partial results obtained from the VFC and LOD assignment stage.

This pipeline can be extended by adding an occlusion-culling stage. Complex scenes such as those in which crowds are needed or those where landscapes are depicted with indigenous vegetation, human elements, buildings and structures can be enhanced. One approach is to perform occlusion culling via OpenGL occlusion queries. Nevertheless, the number of queries needed might not be enough for complex scenes made up of hundreds of thousands of elements. In addition, it requires synchronizing the CPU and the GPU, which might stall the pipeline. Another approach is to put extra cameras in the positions of big structures and perform radar view frustum culling (using the normalized version of the view frustum) and then take the complement of the visible set. This stage can be performed after the behavior stage and before the VFC and LOD assigment stage.

3.8 Acknowledgments

We wish to thank NVIDIA for its kind donation of the GPU used in the experiments.

Bibliography

[Bahnassi 06] Wessam Bahnassi. "AniTextures." In *ShaderX*^{4}*: Advanced Rendering Techniques*. Hingham, MA: Charles River Media, 2006.

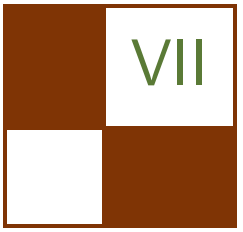
[Dudash 07] B. Dudash. *Skinned Instancing*. NVIDIA Technical Report, 2007.

[Millán et al. 06] Erik Millán, Benjamín Hernández, and Isaac Rudomín. "Large Crowds of Autonomous Animated Characters using Fragment Shaders and Level of Detail." In *ShaderX*^{5}*: Advanced Rendering Techniques*. Higham, MA: Charles River Media, 2006.

[Park and Han 09] Hunki Park and Junghyun Han. "Fast Rendering of Large Crowds Using GPU." In *ICEC '08: Proceedings of the 7th International Conference on Entertainment Computing*, pp. 197–202. Berlin, Heidelberg: Springer-Verlag, 2009.

[Puig Placeres 05] Frank Puig Placeres. "Improved Frustum Culling." In *Game Programming Gems V*. Hingham, MA: Charles River Media, Inc., 2005.

[Rudomín et al. 05] Isaac Rudomín, Erik Millán, and Benjamín Hernández. "Fragment shaders for agent animation using Finite State Machines." *Simulation Modelling Practice and Theory* 13:8 (2005), 741–751.



VII GPGPU

With the constant increase in performance and parallelism of GPUs, there is no doubt that graphic processors have become more general purpose. The introduction of general compute APIs, such as CUDA, OpenCL, and DirectX 11 Compute shaders, have made it possible for modern GPUs to go far beyond the standard processing of triangles and pixels. The latest advances in GPU technologies now allow the implementation of various parallel algorithms, such as AI or physics. With the parallel nature of the GPU, such algorithms can generally run orders of magnitudes faster than their CPU counterparts. This part will cover such general purpose uses of the GPU.

In the first chapter, "2D Distance Field Generation with the GPU," Philip Rideout presents a simple and effective technique to generate distance fields from an image using the OpenCL API. Distance fields have many applications in image processing, real-time rendering, and path-finding algorithms. A distance field is a grayscale bitmap in which each pixel's intensity represents the distance to the nearest contour line in a source image. Rideout explains how such distance fields can be efficiently generated on the GPU and also provides a few examples of how the resulting distance fields can be used in practical applications.

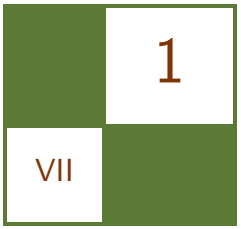
In the second chapter, "Order-Independent Transparency Using Per-Pixel Linked Lists in DirectX 11" by Nicolas Thibieroz, a technique is presented which takes advantage of unordered access views to generate a dynamic link list which can in turn be used to render order-independent translucency. By storing translucent pixels in such a list, they can be sorted as an after process and therefore properly render translucencies regardless of their order.

The third chapter, "Simple and Fast Fluids" by Martin Guay, Fabrice Colin, and Richard Egli, presents a new algorithm that can be used to solve fluid simulations efficiently using compute APIs on the GPU. The article details how the fluid solver works, discusses boundary conditions for single phase flows, and provides a few examples of how such solvers can be used in practical scenarios.

Finally, in "A Fast Poisson Solver for OpenCL using Multigrid Methods" by Sebastien Noury, Samuel Boivin, and Olivier Le Maitre, a novel technique is presented to allow the solving of Poisson partial differential equations using OpenCL. The resolution of Poisson partial differential equations is often needed to solve many techniques in computer graphics, such as fluid dynamics or the merging

and deformation of complex meshes. This article focuses on the presentation of an algorithm that allows for the efficient implementation of a Poisson solver using the GPU.

—Sebastien St-Laurent



2D Distance Field Generation with the GPU

Philip Rideout

Distance fields have many applications in image processing, real-time rendering, and path-finding algorithms. Loosely speaking, a distance field is a grayscale bitmap in which each pixel's intensity represents the distance to the nearest contour line in a source image (see Figure 1.1). The source image often consists of monochrome vector art or text. Distance fields can also be defined in higher dimensions—in the 3D case, the value at each voxel represents distance to the nearest surface. When extended to 3D, many more applications come into play, including facilitation of ray casting and collision detection.

In this chapter we focus on the 2D case. The flat world is much less daunting than 3D, and serves as a good starting point for learning about distance fields. We will focus on the generation of distance fields rather than their application, but will conclude with a brief overview of some rendering techniques that leverage distance fields, including one that enables cheap, high-quality antialiasing.

Perhaps the most classic and commonly-used technique for generating distance fields stems from Per-Erik Danielsson [Danielsson 80]. He describes a method by which pairs of distances are "swept" through an image using a small grid of

Figure 1.1. Fleur-de-lis seed image (left) and its resulting distance field (right).

weights (which he calls a *skeleton*). Danielsson's method is ingenious and much faster than a brute-force technique; however, his algorithm cannot be conveyed succinctly, and is difficult to parallelize on present-day GPU architectures.

Some fascinating and efficient methods for 2D distance field generation on the GPU have been proposed recently [Cao et al. 10] but many of these methods are complex, requiring cunning tricks such as embedding doubly-linked lists within a texture. Here, we give an overview of techniques that are GPU-amenable while still being relatively easy to follow, starting with the simplest (and least accurate) method, which we're calling *Manhattan grassfire*. Although intuitive and easy to implement with OpenGL, this algorithm does not produce accurate results.

After reviewing Manhattan grassfire, we'll introduce a new, more accurate, technique called *horizontal-vertical erosion*, which is also easy to implement using OpenGL. Finally, we'll cover an efficient algorithm proposed by Saito-Toriwaki [Saito and Toriwaki 94]. The nature of their algorithm is amenable to the GPU only when using a compute API rather than a graphics-oriented API. We'll show how we implemented their method using OpenCL.

1.1 Vocabulary

In the context of distance fields, the definition of distance (also known as the *metric*) need not be "distance" in the physical sense that we're all accustomed to.

- *Euclidean metric.* This is the classic definition of distance and corresponds to the physical world.
- *Manhattan metric.* The sum of the axis-aligned horizontal and vertical distances between two points. As Pythagoras taught us, city block distance is not equivalent to Euclidean distance. However, it tends to be much easier to compute.
- *Chessboard metric.* Rather than summing the horizontal and vertical distances, take their maximum. This is the minimum number of moves a king needs when traveling between two points on a chessboard. Much like the Manhattan metric, chessboard distance tends to be easier to compute than true Euclidean distance.
- *Squared Euclidean metric.* This is a distance field where each value is squared distance rather than true distance. This is good enough for many applications, and easier to compute. It also serves as a convenient intermediary step when computing true Euclidean distance.
- *Seed image.* Ordinarily this isn't thought of as a distance metric; it's a binary classification consisting of *object pixels* and *background pixels*. In this article, object pixels are depicted in black and background pixels are white.

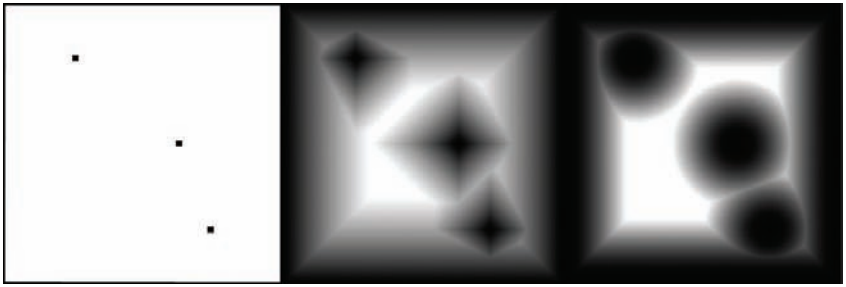


Figure 1.2. Seed image (left), Manhattan distance (middle), Euclidean distance (right).

See Figure 1.2 for an example of a seed image and a comparison of Manhattan and Euclidean metrics.

It also helps to classify the generation algorithms that are amenable to the GPU:

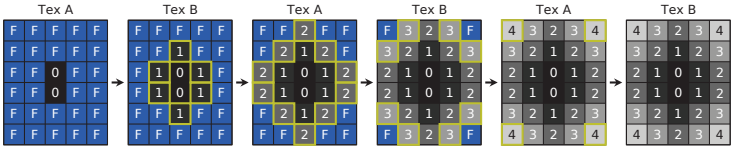
- *Grassfire.* These types of algorithms incrementally flood the boundaries of the vector art outward, increasing the distance values along the way. Once a pixel has been set, it never changes. The algorithm's termination condition is that all background pixels have been set.
- *Erosion.* Much like grassfire methods, erosion algorithms iteratively manipulate the entire image. Unlike grassfire, pixel values are constantly readjusting until an equilibrium state has been reached.
- *Scanning.* Instead of generating a succession of new images, scanning methods "sweep" through the seed image, propagating values horizontally or vertically. Scanning techniques are less amenable to OpenGL, but are often parallelizable using OpenCL or CUDA.
- *Voronoi.* Distance fields are closely related to Voronoi maps, which are classically generated on the GPU by drawing z-aligned cones into the depth buffer. Alternatively, cone tessellation can be avoided by splatting soft particles with a blending equation of `min(source, dest)`. Voronoi-based techniques tend to be raster-heavy, and are more amenable to source images that contain clouds of discrete points (much like Figure 1.2) rather than source images that have continuous outlines (like the Fleur-de-Lis shape). *Jump flooding* is an interesting Voronoi-based technique presented in [Rong and Tan 06].

Grassfire and erosion techniques are typically implemented with a graphics API using an image-processing technique called *ping-ponging*. The fragment shader cannot read values back from the render target, so two image surfaces are

created: surface **A** and surface **B**. On even-numbered passes, surface **A** is used as a source texture and surface **B** is used as the render target. On odd-numbered passes, surface **B** is the source texture and surface **A** is the render target.

1.2 Manhattan Grassfire

The following diagram illustrates a well-known method for computing the Manhattan metric using a 4 bits-per-pixel surface. The left-most image is the seed image, and the right-most image contains the final distance field. Blue cells represent background pixels and contain a value of `0xF`. Note that the last two images are identical; the last pass is intentionally redundant to allow an occlusion query to signal termination (more on this later).



The fragment shader (for OpenGL 3.0 and above) used in each rendering pass is shown in Listing 1.1. This shader assumes that the render target and source texture have formats that are single-component, 8-bit unsigned integers.

```
out uint FragColor;
uniform usampler2D Sampler;

void main()
{
    ivec2 coord = ivec2(gl_FragCoord.xy);

    uint color = texelFetch(Sampler,coord,0).r;
    if (color != 255u)
        discard;

    uint n = texelFetchOffset(Sampler,coord,0,ivec2(0,-1)).r;
    uint s = texelFetchOffset(Sampler,coord,0,ivec2(0,+1)).r;
    uint e = texelFetchOffset(Sampler,coord,0,ivec2(+1,0)).r;
    uint w = texelFetchOffset(Sampler,coord,0,ivec2(-1,0)).r;

    FragColor = min(n,min(s,min(e,w)));
    if (FragColor < 254u)
        FragColor++;
}
```

Listing 1.1. Grassfire fragment shader.

To summarize the shader: if the current pixel is not a background pixel, it can be skipped because it has already been filled. Otherwise, find the minimum value from the neighboring pixels in the four cardinal directions (`n`, `s`, `e`, `w`). To enhance this shader to compute a chessboard metric, simply add four new texture lookups for the diagonal neighbors (`ne`, `nw`, `se`, `sw`).

The grassfire algorithm is easy to implement, but it's not obvious how to detect when a sufficient number of passes has been completed. In the worst case, the number of passes is `max(width, height)`. In practice, the number of passes is much fewer.

The occlusion query capabilities in modern graphics hardware can help. In Listing 1.1, we issue a `discard` statement for non-background pixels. You might wonder why we didn't do this instead:

```
if (color != 255u) {
    FragColor = color;
    return;
}
```

Using `discard` instead of `return` is crucial; it allows us to leverage an occlusion query to terminate the image processing loop. If all pixels are discarded, then no change occurs, and the algorithm is complete.

One consequence of using `discard` in this way is that some pixels in destination surface are left uninitialized. To fix this, we need to blit the entire source texture before running the erosion shader. Luckily this is a fast operation on modern GPUs. See Listing 1.2 for the complete image processing loop.

```
bool done = false;
int pass = 0;
while (!done) {

    // Swap the source & destination surfaces and bind them
    Swap(Source, Dest);
    glBindFramebuffer(GL_FRAMEBUFFER, Dest.RenderTarget);
    glBindTexture(GL_TEXTURE_2D, Source.TextureHandle);

    // Copy the entire source image to the target
    glUseProgram(BlitProgram);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);

    // Execute the grassfire shader and measure the pixel count
    glUseProgram(GrassfireProgram);
    glBeginQuery(GL_SAMPLES_PASSED, QueryObject);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    glEndQuery(GL_SAMPLES_PASSED);

    // If all pixels were discarded, we're done
    GLuint count = 0;
```

```
    glGetQueryObjectuiv(QueryObject,GL_QUERY_RESULT,&count);
    done = (count == 0) || (pass++ > MaxPassCount);
}
```

Listing 1.2. C algorithm for Manhattan grassfire.

Note that Listing 1.2 also checks against the `MaxPassCount` constant for loop termination. This protects against an infinite loop in case an error occurs in the fragment shader or occlusion query.

1.3 Horizontal-Vertical Erosion

Although intuitive, the method presented in the previous section is only marginally useful, because it computes distance according to a city block metric. In this section, we present a new technique that generates distance according to a squared Euclidean metric. The algorithm consists of two separate image-processing loops. The first loop makes a succession of horizontal transformations and the second loop makes a succession of vertical transformations. At each pass, an odd integer offset is applied to the propagated distance values (β in Figures 1.3 and 1.4, which illustrate the process on a 4-bit surface). This is similar to a parallel method proposed by [Lotufo and Zampirolli 01].

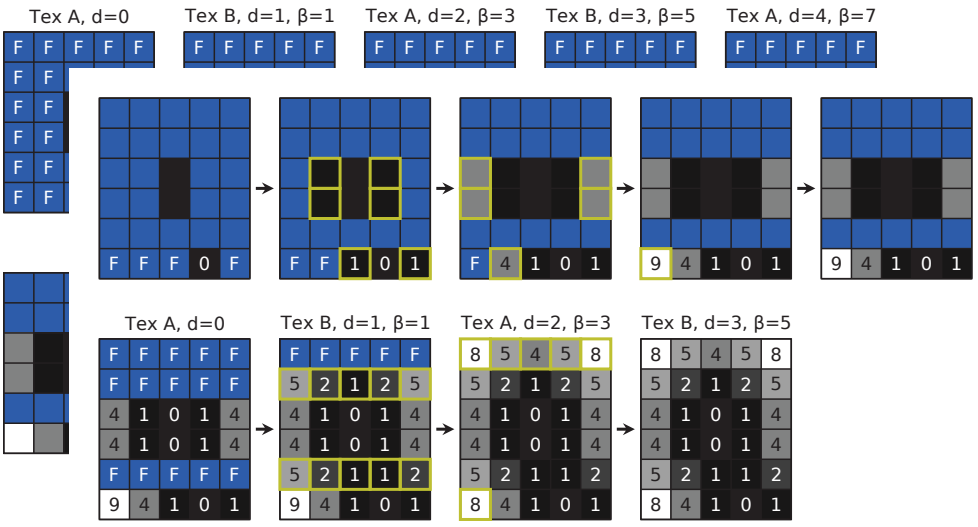


Figure 1.4. Vertical erosion.

You might be wondering why this is erosion rather than grassfire, according to our terminology. In Figure 1.4, notice that a **9** changes into an **8** during the second pass. Since a nonbackground pixel changes its value, this is an erosion-based method.

The fragment shader for the horizontal stage of processing is shown next.

```
out uint FragColor;
uniform usampler2D Sampler;
uniform uint Beta;
uniform uint MaxDistance;

void main()
{
    ivec2 coord = ivec2(gl_FragCoord.xy);

    uint A = texelFetch(Sampler,coord,0).r;
    uint e = texelFetchOffset(Sampler,coord,0,ivec2(+1,0)).r;
    uint w = texelFetchOffset(Sampler,coord,0,ivec2(-1,0)).r;
    uint B = min(min(A,e+Beta),w+Beta);

    if (A == B || B > MaxDistance)
        discard;

    FragColor = B;
}
```

Listing 1.3. Erosion fragment shader.

Background pixels are initially set to "infinity," (i.e., the largest possible value allowed by the bit depth). Since the shader discards pixels greater than the application-defined `MaxDistance` constant, it effectively clamps the distance values. We'll discuss the implications of clamping later in this chapter.

To create the shader for the vertical pass, simply replace the two East-West offsets (+1,0) and (-1,0) in Listing 1.3 with North-South offsets (0,+1) and (0,-1).

To give the erosion shaders some context, the C code is shown next.

```
GLuint program = HorizontalProgram;
for (int d = 1; d < MaxPassCount; d++) {

    // Swap the source & destination surfaces and bind them
    Swap(Source,Dest);
    glBindFramebuffer(GL_FRAMEBUFFER,Dest.RenderTarget);
    glBindTexture(GL_TEXTURE_2D,Source.TextureHandle);

    // Copy the entire source image to the destination surface
    glUseProgram(BlitProgram);
```

```
        glDrawArrays(GL_TRIANGLE_FAN,0,4);

        // Execute the erosion shader and measure the pixel count
        glUseProgram(program);
        glUniform1ui(Uniforms.Beta, 2*d-1);
        glUniform1ui(Uniforms.MaxDistance, 65535);
        glBeginQuery(GL_SAMPLES_PASSED,QueryObject);
        glDrawArrays(GL_TRIANGLE_FAN,0,4);
        glEndQuery(GL_SAMPLES_PASSED);

        // If all pixels were discarded, we're done with this stage
        GLuint count = 0;
        glGetQueryObjectuiv(QueryObject,GL_QUERY_RESULT,&count);
        if (count == 0) {
            if (program == HorizontalProgram) {
                program = VerticalProgram;
                d = 0;
            } else {
                break;
            }
        }
    }
```

Listing 1.4. C algorithm for horizontal-vertical erosion.

Applying odd-numbered offsets at each pass might seem less intuitive than the Manhattan technique, but the mathematical explanation is simple. Recall that the image contains squared distance, d^2. At every iteration, the algorithm fills in new distance values by adding an offset value β to the previous distance. Expressing this in the form of an equation, we have

$$d^2 = (d-1)^2 + \beta.$$

Solving for β is simple algebra:

$$\beta = 2 * d - 1;$$

therefore, β iterates through the odd integers.

1.4 Saito-Toriwaki Scanning with OpenCL

Saito and Toriwaki developed another technique that consists of a horizontal stage of processing followed by a vertical stage [Saito and Toriwaki 94]. Interestingly, the horizontal stage of processing is not erosion-based and is actually $O(n)$. Their method is perhaps one of the best ways to generate squared Euclidean distance on the CPU, because it's fast and simple to implement. Their algorithm is also fairly amenable to the GPU, but only when using a compute API such as OpenCL rather than a purely graphics-oriented API.

Several researchers have improved the worst-case efficiency of the Saito-Toriwaki algorithm, but their methods often require additional data structures, such as a stack, and their complexity makes them less amenable to the GPU. (For an excellent survey of techniques, see [Fabbri et al. 08].) We found the original Saito-Toriwaki algorithm to be easy to implement and parallelize. At a high level, their method can be summarized in two steps:

1. Find the one-dimensional distance field of each row. This can be performed efficiently in two passes as follows:

   - First, crawl rightward and increment a counter along the way, resetting the counter every time you cross a contour line. Write the counter's value into the destination image along the way. After the entire row is processed, the seed image can be discarded.
   - Next, crawl leftward, similarly incrementing a counter along the way and writing the values into the destination image. When encountering an existing value in the destination image that's less than the current counter, reset the counter to that value.

   In code, the operations performed on a single row can be expressed as follows:

```
// Rightward Pass
d = 0;
for (x = 0; x < Width; x++) {
    d = seed[x] ? 0 : d+1;
    destination[x] = d;
}

// Leftward Pass
d = 0;
for (x = Width-1; x >= 0; x--) {
    d = min(d+1, destination[x]);
    destination[x] = d;
}
```

2. In each vertical column, find the minimum squared distance of each pixel, using only the values computed in Step 1 as input. A brute-force way of doing this would be as follows:

```
for (y1 = 0; y1 < height; y1++) {
    minDist = INFINITY;
    for (y2 = 0; y2 < height; y2++) {
        d = destination[y2];
        d = (y1 - y2) * (y1 - y2) + d*d;
        minDist = min(minDist, d);
    }
```

```
            destination[y1] = minDist;
        }
```

Note the expensive multiplications in the vertical pass. They can be optimized in several ways:

- The $d * d$ operation can be pulled out and performed as a separate pass on the entire image, potentially making better use of GPU parallelization.
- The $d * d$ operation can be eliminated completely by leveraging the β trick that we described in our erosion-based method. In this way, the horizontal pass would track squared distance from the very beginning.
- The $(y1 - y2)^2$ operation can be replaced with a lookup table because $|y1 - y2|$ is a member of a relatively small set of integers.

In practice, we found that these multiplications were not very damaging since GPUs tend to be extremely fast at multiplication.

For us, the most fruitful optimization to the vertical pass was splitting it into downward and upward passes. Saito and Toriwaki describe this in detail, showing how it limits the range of the inner loop to a small region of interest.

Figure 1.5. Seed image, rightward, leftward, downward, upward (top to bottom).

This optimization doesn't help much in worst-case scenarios, but it provides a substantial boost for most images. To see how to split up the vertical pass, see the accompanying sample code. The step-by-step process of the Saito-Toriwaki transformation is depicted in Figure 1.5.

In reviewing the OpenCL implementation of the Saito-Toriwaki algorithm, we first present a naive but straightforward approach. We then optimize our implementation by reducing the number of global memory accesses and changing the topology of the OpenCL kernels.

1.4.1 OpenCL Setup Code

Before going over the kernel source, let's first show how the CPU-side code sets up the OpenCL work items and launches them. For the sake of brevity, we've omitted much of the error-checking that would be expected in production code (see Listing 1.5).

```
void RunKernels(cl_uchar* inputImage, cl_ushort* outputImage,
                cl_platform_id platformId, const char* source)
{
    size_t horizWorkSize[] = {IMAGE_HEIGHT};
    size_t vertWorkSize[] = {IMAGE_WIDTH};
    cl_context context;
    cl_mem inBuffer, outBuffer, scratchBuffer;
    cl_program program;
    cl_kernel horizKernel, vertKernel;
    cl_command_queue commandQueue;
    cl_device_id deviceId;

    // Create the OpenCL context
    clGetDeviceIDs(platformId, CL_DEVICE_TYPE_GPU, 1,
       &deviceId, 0);
    context = clCreateContext(0, 1, &deviceId, 0, 0, 0);

    // Create memory objects
    inBuffer = clCreateBuffer(context,
                CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                IMAGE_WIDTH * IMAGE_HEIGHT, inputImage, 0);
    outBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE,
                IMAGE_WIDTH * IMAGE_HEIGHT * 2, 0, 0);

    // Load and compile the kernel source
    program = clCreateProgramWithSource(context, 1,
                                        &source, 0, 0);
    clBuildProgram(program, 0, 0, "-cl-fast-relaxed-math", 0, 0);

    // Set up the kernel object for the horizontal pass
    horizKernel = clCreateKernel(program, "horizontal", 0);
    clSetKernelArg(horizKernel, 0, sizeof(cl_mem), &inBuffer);
    clSetKernelArg(horizKernel, 1, sizeof(cl_mem), &outBuffer);
```

```
    // Set up the kernel object for the vertical pass
    vertKernel = clCreateKernel(program, "vertical", 0);
    clSetKernelArg(vertKernel, 0, sizeof(cl_mem), &outBuffer);

    // Execute the kernels and read back the results
    commandQueue = clCreateCommandQueue(context, deviceId, 0, 0);
    clEnqueueNDRangeKernel(commandQueue, horizKernel, 1, 0,
                           horizWorkSize, 0, 0, 0, 0);
    clEnqueueNDRangeKernel(commandQueue, vertKernel, 1, 0,
                           vertWorkSize, 0, 0, 0, 0);
    clEnqueueReadBuffer(commandQueue, outBuffer, CL_TRUE, 0,
                        IMAGE_WIDTH * IMAGE_HEIGHT * 2,
                        outputImage, 0, 0, 0);
}
```

Listing 1.5. OpenCL Saito -Toriwaki algorithm.

Listing 1.5 uses OpenCL memory buffers rather than OpenCL image objects. This makes the kernel code a bit easier to follow for someone coming from a CPU background. Since we're not leveraging the texture filtering capabilities of GPUs anyway, this is probably fine in practice.

Also note that we're using a seed image that consists of 8-bit unsigned integers, but our target image is 16 bits. Since we're generating *squared* distance, using only 8 bits would result in very poor precision. If desired, a final pass could be tacked on that takes the square root of each pixel and generates an 8-bit image from that.

1.4.2 Distance Clamping

The finite bit depth of the target surface leads to an important detail in our implementation of the Saito-Toriwaki algorithm: distance clamping. Rather than blindly incrementing the internal distance counter as it marches horizontally, our implementation keeps the distance clamped to a maximum value like this:

```
ushort nextDistance = min(254u, distance) + 1u
```

Even though the target is 16 bit, we clamp it to 255 during the horizontal scan because it gets squared in a later step. Note that distance clamping results in an interesting property:

> If distances are clamped to a maximum value of x, then any two seed pixels further apart than x have no effect on each other in the final distance field.

We'll leverage this property later. For some applications, it's perfectly fine to clamp distances to a very small value. This can dramatically speed up the generation algorithm, as we'll see later.

1.4.3 OpenCL Kernels: Naïve and Optimized Approaches

Listing 1.6 is the complete listing of the horizontal kernel used in our naive implementation. For simplicity, this kernel operates on a fixed-width image. In practice, you'll want to pass in the width as an argument to the kernel.

```
kernel void horizontal(global const uchar* indata,
   global ushort* outdata)
{
    const int y = get_global_id(0);
    global const uchar* source = indata + y * IMAGE_WIDTH;
    global ushort* target = outdata + y * IMAGE_WIDTH;
    uint d;

    // Rightward pass
    d = 0;
    for (int x = 0; x < IMAGE_WIDTH; x++) {
        ushort next = min(254u, d) + 1u;
        d = source[x] ? 0u : next;
        target[x] = d;
    }

    // Leftward pass
    d = 0;
    for (int x = IMAGE_WIDTH - 1; x >= 0; x--) {
        ushort next = min(254u, d) + 1u;
        d = min(next, target[x]);
        target[x] = d;
    }

    // Square the distances
    for (int x = 0; x < IMAGE_WIDTH; x++) {
        target[x] = target[x]  * target[x];
    }
}
```

Listing 1.6. Naïve horizontal kernel.

The biggest issue with the code in Listing 1.6 is the numerous accesses to shared memory. Specifically, the number of reads and writes to the `target` pointer should be reduced. On present-day GPU architectures, each access to global memory incurs a huge performance hit.

The obvious way to reduce these high-latency accesses is to change the kernel so that it copies the entire row into local memory and performs all the necessary processing locally. As a final step, the kernel copies the results back out to the shared memory. This strategy would require enough local memory to accommodate an entire row in the image, which in our case exceeds the hardware cache size.

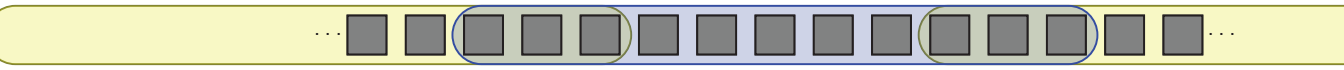


Figure 1.6. OpenCL work item for the horizontal pass (blue), with two overlapping neighbors (yellow).

To reduce the need for generous amounts of local storage, we can break up each row into multiple sections, thus shrinking the size of each OpenCL work item. Unfortunately, operating on a narrow section of the image can produce incorrect results, since contour lines outside the section are ignored.

This is where we leverage the fact that far-apart pixels have no effect on each other when clamping the distance field. The middle part of each work item will produce correct results since it's far away from neighboring work items. We'll use the term *margin* to label the incorrect regions of each work item. By overlapping the work items and skipping writes for the values in the margin, the incorrect regions of each work item are effectively ignored (see Figures 1.6 and 1.7). Note that tighter distance clamping allows for smaller margin size, resulting in better parallelization.

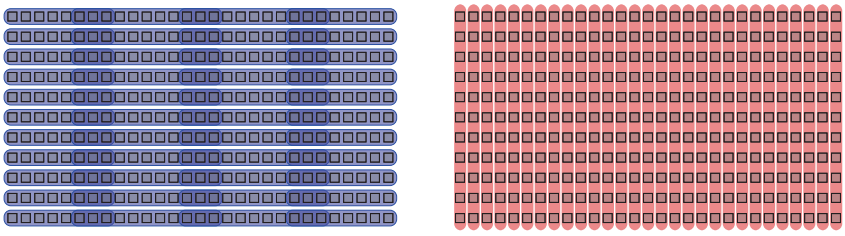


Figure 1.7. OpenCL topology for the horizontal and vertical kernels.

We now need to set up the work groups for the horizontal pass using a two-dimensional arrangement (see Listing 1.7); changed lines are highlighted. The `SPAN` constant refers to the length of each work item, not including the throw-away margins.

```
void RunKernels(cl_uchar* inputImage, cl_ushort* outputImage,
                cl_platform_id platformId,
                const char* kernelSource)
{
    size_t horizWorkSize[] = {IMAGE_WIDTH / SPAN, IMAGE_HEIGHT};
    size_t vertWorkSize[] = {IMAGE_WIDTH};
```

```
    // function body is same as before -- see Listing 1.5

    // Execute the kernels and read back the results
    commandQueue = clCreateCommandQueue(context, deviceId, 0, 0);
    clEnqueueNDRangeKernel(commandQueue, horizKernel, 2, 0,}
                           horizWorkSize, 0, 0, 0, 0);}
    clEnqueueNDRangeKernel(commandQueue, vertKernel, 1, 0,
                           vertWorkSize, 0, 0, 0, 0);
    clEnqueueReadBuffer(commandQueue, outBuffer, CL_TRUE, 0,
                        IMAGE_WIDTH * IMAGE_HEIGHT * 2,
                        outputImage, 0, 0, 0);
}
```

Listing 1.7. Optimized Saito-Toriwaki algorithm.

Listing 1.8 is the listing for the new kernel code. Instead of looping between 0 and `WIDTH`, we now perform processing between `Left` and `Right`, which are determined from the value returned by `get_global_id(0)`. You'll also notice the `InnerLeft` and `InnerRight` constants; these mark the portion of the work item that actually gets written out.

```
kernel void horizontal(global const uchar* indata,
                       global ushort* outdata)
{
    const int y = get_global_id(1);
    global const uchar* source = indata + y * IMAGE_WIDTH;
    global ushort* target = outdata + y * IMAGE_WIDTH;
    uchar scratch[MARGIN + SPAN + MARGIN];
    uint d;

    const int InnerLeft = SPAN * get_global_id(0);
    const int InnerRight = min(IMAGE_WIDTH, InnerLeft + SPAN);
    const int Left = max(0, InnerLeft - MARGIN);
    const int Right = min(IMAGE_WIDTH, InnerRight + MARGIN);

    // Rightward pass
    d = 0;
    for (int x = Left; x < Right; x++) {
        ushort next = min(254u, d) + 1u;
        d = source[x] ? 0u : next;
        scratch[x - Left] = d;
    }

    // Leftward pass
    d = 0;
    for (int x = Right - 1; x >= Left; x--) {
        ushort next = min(254u, d) + 1u;
        d = min(next, (ushort) scratch[x - Left]);
        scratch[x - Left] = d;
```

```
    }

    // Square the distances and write them out to shared memory
    for (int x = InnerLeft; x < InnerRight; x++) {
        target[x] = scratch[x - Left] * scratch[x - Left];
    }
}
```

Listing 1.8. Optimized horizontal kernel.

The only remaining piece is the kernel for the vertical pass. Recall the code snippet we presented earlier that described an $O(n^2)$ algorithm to find the minimum distances in a column. By splitting the algorithm into downward and upward passes, Saito and Toriwaki show that the search area of the inner loop can be narrowed to a small region of interest, thus greatly improving the best-case efficiency. See this book's companion source code for the full listing of the vertical kernel.

Due to the high variability from one type of GPU to the next, we recommend that readers experiment to find the optimal OpenCL kernel code and topology for their particular hardware.

Readers may also want to experiment with the image's data type (floats versus integers). We chose integers for this article because squared distance in a grid is, intuitively speaking, never fractional. However, keep in mind that GPUs are floating-point monsters! Floats and half-floats may provide better results with certain architectures. It suffices to say that the implementation presented in this article is by no means the best approach in every circumstance.

1.5 Signed Distance with Two Color Channels

Regardless of which generation algorithm is used, or which distance metric is employed, a choice exists of generating *signed distance* or *unsigned distance*. The

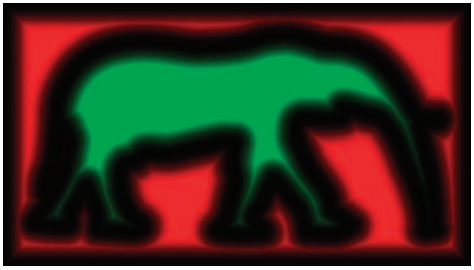


Figure 1.8. Signed distance in two color channels.

former generates distances for object pixels in addition to background pixels. Object pixels have negative distance while background pixels have positive distance.

It's easy to extend any technique to account for signed distance by simply inverting the seed image and applying the algorithm a second time. We found it convenient to use an unsigned, integer-based texture format, and added a second color channel to the image for the negative values. In Figure 1.8, we depict a signed distance field where the red channel contains positive distance and the green channel contains negative distance.

In the case of the horizontal-vertical erosion technique presented earlier, we can modify the fragment shader to operate on two color channels simultaneously, thus avoiding a second set of passes through the image for negative distance. Listing 1.9 shows the new fragment shader.

```
out uvec2 FragColor;
uniform usampler2D Sampler;
uniform uint Beta;

void main()
{
    ivec2 coord = ivec2(gl_FragCoord.xy);

    uvec2 A = texelFetch(Sampler,coord,0).rg;
    uvec2 e = texelFetchOffset(Sampler,coord,0,ivec2(+1,0)).rg;
    uvec2 w = texelFetchOffset(Sampler,coord,0,ivec2(-1,0)).rg;
    uvec2 B = min(min(A,e+Beta),w+Beta);

    if (A == B)
        discard;

    FragColor = B;
}
```

Listing 1.9. Erosion fragment shader for signed distance.

For most applications it's desirable to make a final transformation that normalizes the two-channel distance into a simple grayscale image. The final square-root transformation can also be performed at this time. The following fragment shader makes this final pass:

```
varying vec2 TexCoord;
uniform sampler2D Sampler;
uniform float Scale;

void main()
{
    vec2 D = sqrt(texture2D(Sampler, TexCoord).rg);
```

```
    float L = 0.5 + Scale * (D.r - D.g);
    gl_FragColor = vec4(L);
}
```

If a distance field is normalized in this way, a value of 0.5 indicates the center of a contour in the seed image.

1.6 Distance Field Applications

Several interesting uses of distance fields for 2D rendering have been popularized by Chris Green of Valve software [Green 07], who builds on work by [Qin et al. 06]. Leveraging alpha testing (as opposed to alpha *blending*) in conjunction with distance fields enables high-quality magnification of vector art. This is useful on low-end hardware that doesn't support shaders, or when using a fixed-function graphics API like OpenGL ES 1.1.

If shaders are supported, then high-quality anisotropic antialiasing can be achieved using inexpensive bilinear filtering against a low-resolution distance field. (This assumes that the shading language supports derivatives.) Additionally, special effects are easily rendered, including outlines, drop shadows, and glow effects. We'll briefly cover some of these effects in the following sections, using fragment shaders written for OpenGL 3.0.

1.6.1 Antialiasing

Both of the source textures in Figure 1.9 are only 128×32; it's obvious that rendering with the aid of a distance field can provide much better results.

Because the gradient vector at a certain pixel in the distance field gives the direction of maximum change, it can be used as the basis for antialiasing. The `fwidth` function in GLSL provides a convenient way to obtain the rate of change of the input value at the current pixel. In our case, large values returned from `fwidth` indicate a far-off camera, while small values indicate large magnification.

Recall that a lookup value of 0.5 represents the location of the contour line. We compute the best alpha value for smoothing by testing how far the current pixel is from the contour line. See Listing 1.10 for our antialiasing shader.

Figure 1.9. Bilinear filtering (left). Magnification using a distance field (right).

```
in vec2 TexCoord;
out vec4 FragColor;
uniform sampler2D Sampler;
void main()
{
    float D = texture(Sampler, TexCoord).x;
    float width = fwidth(D);
    float A = 1.0 - smoothstep(0.5 - width, 0.5 + width, D);
    FragColor = vec4(0, 0, 0, A);
}
```

Listing 1.10. Fragment shader for antialiasing with a distance field.

Figure 1.10. Outline effect

1.6.2 Outlining

Creating an outline effect such as the one depicted in Figure 1.10 is quite simple when using a signed distance field for input. Note that there are *two* color transitions that we now wish to antialias: the transition from the fill color to the outline color, and the transition from the outline color to the background color. The following fragment shader shows how to achieve this; the `Thickness` uniform is the desired width of the outline.

```
in vec2 TexCoord;
out vec4 FragColor;
uniform sampler2D Sampler;
uniform float Thickness;

void main()
{
    float D = texture(Sampler, TexCoord).x;
    float W = fwidth(D);
    float T0 = 0.5 - Thickness;
    float T1 = 0.5 + Thickness;
```

```
    if (D < T0) {
        float A = 1.0 - smoothstep(T0-W, T0, D);
        FragColor = vec4(A, A, A, 1);
    } else if (D < T1) {
        FragColor = vec4(0, 0, 0, 1);
    } else {
        float A = 1.0 - smoothstep(T1, T1+W, D);
        FragColor = vec4(0, 0, 0, A);
    }
}
```

1.6.3 Psychedelic Effect

We conclude the chapter with a fun (but less practical) two-dimensional effect (see Figure 1.11). This effect is simple to achieve by mapping distance to hue, and performing HSV-to-RGB conversion in the fragment shader. Animating an offset value (the `Animation` uniform) creates a trippy 70's effect.

```
in vec2 TexCoord;
out vec4 FragColor;
uniform sampler2D Sampler;
uniform float Animation;

void main()
{
    float D = texture(Sampler, TexCoord).x;
    float W = fwidth(D);
    float H = 2.0 * float(D - 0.5);
    float A = smoothstep(0.5 - W, 0.5 + W, D);
    hue = fract(H + Animation);
    FragColor = vec4(A * HsvToRgb(H, 1.0, 1.0), 1.0);
}
```

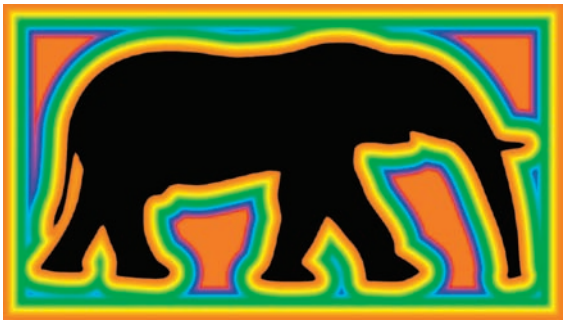


Figure 1.11. Psychedelic effect

Bibliography

[Cao et al. 10] Thanh-Tung Cao, Ke Tang, Anis Mohamed, and Tiow-Seng Tan. "Parallel Banding Algorithm to Compute Exact Distance Transform with the GPU." In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 83–90. New York: ACM, 2010.

[Danielsson 80] P. E. Danielsson. "Euclidean Distance Mapping." *Computer Graphics and Image Processing* 14:3 (1980), 227–248.

[Fabbri et al. 08] Ricardo Fabbri, Luciano da F. Costa, Julio C. Torelli, and Odemir M. Bruno. "2D Euclidean Distance Transform Algorithms: A Comparative Survey." *ACM Computing Surveys* 40:1 (2008), 2:1–2:44.

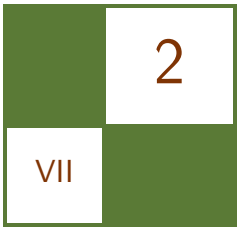
[Green 07] Chris Green. "Improved Alpha-Tested Magnification for Vector Textures and Special Effects." In *SIGGRAPH '07: ACM SIGGRAPH 2007 Courses*, pp. 9–18. New York: ACM, 2007.

[Lotufo and Zampirolli 01] Roberto de Alencar Lotufo and Francisco A. Zampirolli. "Fast Multidimensional Parallel Euclidean Distance Transform Based on Mathematical Morphology." In *SIBGRAPI '01: Proceedings of the 14th Brazilian Symposium on Computer Graphics and Image Processing*, pp. 100–105. Washington, DC: IEEE Computer Society, 2001.

[Qin et al. 06] Zheng Qin, Michael D. McCool, and Craig S. Kaplan. "Real-Time Texturemapped Vector Glyphs." In *Symposium on Interactive 3D Graphics and Games*, pp. 125–132. New York: ACM Press, 2006.

[Rong and Tan 06] Guodong Rong and Tiow-Seng Tan. "Jump Flooding: An Efficient and Effective Communication on GPU." pp. 185–192. Hingham, MA: Charles River Media, 2006.

[Saito and Toriwaki 94] Toyofumi Saito and Jun-Ichiro Toriwaki. "New algorithms for Euclidean Distance Transformation of an n-Dimensional Digitized Picture with Applications." *Pattern Recognition* 27:11 (1994), 1551–1565.



Order-Independent Transparency using Per-Pixel Linked Lists

Nicolas Thibieroz

2.1 Introduction

Order-independent transparency (OIT) has been an active area of research in real-time computer graphics for a number of years. The main area of research has focused on how to effect fast and efficient back-to-front sorting and rendering of translucent fragments to ensure visual correctness when order-dependent blending modes are employed. The complexity of the task is such that many real-time applications have chosen to forfeit this step altogether in favor of simpler and faster alternative methods such as sorting per object or per triangle, or simply falling back to order-independent blending modes (e.g., additive blending) that don't require any sorting [Thibieroz 08]. Different OIT techniques have previously been described (e.g., [Everitt 01], [Myers 07]) and although those techniques succeed in achieving correct ordering of translucent fragments, they usually come with performance, complexity, or compatibility shortcomings that make their use difficult for real-time gaming scenarios.

This chapter presents an OIT implementation relying on the new features of the DirectX 11 API from Microsoft.[1] An overview of the algorithm will be presented first, after which each phase of the method will be explained in detail. Sorting, blending, multisampling, anti-aliasing support, and optimizations will be treated in separate sections before concluding with remarks concerning the attractiveness and future of the technique.

2.2 Algorithm Overview

The OIT algorithm presented in this chapter shares some similarities with the A-buffer algorithm [Carpenter 84], whereby translucent fragments are stored in

[1] "Microsoft DirectX" is a registered trademark.

buffers for later rendering. Our method uses linked lists [Yang 10] to store a list of translucent fragments for each pixel. Therefore, every screen coordinate in the render viewport will contain an entry to a unique per-pixel linked list containing all translucent fragments at that particular location.

Prior to rendering translucent geometry, all opaque and transparent (alpha-tested) models are rendered onto the render target viewport as desired. Then the OIT algorithm can be invoked to render corrected-ordered translucent fragments.

The algorithm relies on a two-step process.

1. The first step is the creation of per-pixel linked lists whereby the translucent contribution to the scene is entirely captured into a pair of buffers containing the head pointers and linked lists nodes for all translucent pixels.

2. The second step is the traversal of per-pixel linked lists to fetch, sort, blend and finally render all pixels in the correct order onto the destination render viewport.

2.3 DirectX 11 Features Requisites

DirectX 11 has introduced new features that finally make it possible to create and parse concurrent linked lists on the GPU.

2.3.1 Unordered Access Views

Unordered access view (UAV) is a special type of resource view that can be bound as output to a pixel or compute shader to allow the programmer to write data at arbitrary locations. The algorithm uses a pair of UAVs to store per-pixel linked lists nodes and head pointers.

2.3.2 Atomic Operations

The creation of per-pixel linked lists also requires a way to avoid any contention when multiple pixel shader invocations perform memory operations into a buffer. Indeed such read/modify/write operations must be guaranteed to occur atomically for the algorithm to work as intended. DirectX 11 supports a set of `Interlocked*()` Shader Model 5.0 instructions that fulfill this purpose. Such atomic operation will be used to keep track of the head pointer address when creating the per-pixel linked lists.

2.3.3 Hardware Counter Support on UAV

A less well-known feature of DirectX 11 allows the creation of a built-in hardware counter on buffer UAVs. This counter is declared by specifying the `D3D11_BUFFER_UAV_FLAG_COUNTER` flag when generating a UAV for the intended

buffer. The programmer is given control of the counter via the following two Shader Model 5.0 methods:

```
uint <Buffer>.IncrementCounter();
uint <Buffer>.DecrementCounter();
```

Hardware counter support is used to keep track of the offset at which to store the next linked list node.

While hardware counter support is not strictly required for the algorithm to work, it enables considerable performance improvement compared to manually keeping track of a counter via a single-element buffer UAV.

2.3.4 Early Depth/Stencil Rejection

Graphics APIs like OpenGL and Direct3D specify that the depth/stencil test be executed *after* the pixel shader stage in the graphics pipeline. A problem arises when the pixel shader outputs to UAVs because UAVs may be written into the shader even though the subsequent depth/stencil test actually discards the pixel, which may not be the intended behavior of the algorithm. Shader Model 5.0 allows the `[earlydepthstencil]` keyword to be declared in front of a pixel shader function to indicate that the depth/stencil test is to be explicitly performed *before* the pixel shader stage, allowing UAV writes to be carried out only *if the depth/ stencil test succeeds first.* This functionality is important for the algorithm presented in this chapter, since only *visible* translucent fragments need storing into the per-pixel linked lists.

2.3.5 SV_COVERAGE Pixel Shader Input

DirectX 11 allows `SV_COVERAGE` to be declared as an input to the pixel shader stage. `SV_COVERAGE` contains a bit mask of all samples that are covered by the current primitive. This information is used by this OIT technique when multi-sampling antialiasing (MSAA) is enabled.

2.3.6 Per-sample Pixel Shader Execution

DirectX 11 allows the pixel shader stage to execute *per sample* (as opposed to *per pixel*) when MSAA is enabled. This functionality will be exploited to allow MSAA support with our OIT technique.

2.4 Head Pointer and Nodes Buffers

The algorithm builds a reverse linked list for each pixel location in the target viewport. The linked list *head pointer* is the address of the first element in the

linked list. The linked list *nodes* are the individual elements of the linked list that contain fragment data as well as the address of the next node.

2.4.1 Head Pointer Buffer

The algorithm allocates a screen-sized buffer of type `DXGI_FORMAT_R32_UINT` that contains the address of the head pointer for every 2D coordinate on the screen.

Despite the resource having the same dimensions as the render viewport, the declaration must employ the buffer type because atomic operations are not supported on Texture2D resources. Therefore an extra step will be required in the shader wherein a 2D screen-coordinate location is converted to a byte-linear address:

```
uint uLinearAddressInBytes = 4 * ( ScreenPos.y*RENDERWIDTH +
 ScreenPos.x );
```

The head pointer buffer is initialized to a "magic" value of 0xFFFFFFFF, indicating that the linked list is empty to start with. In effect, an address of 0xFFFFFFFF indicates that no more nodes are available (i.e., the end of the list has been reached).

The term *reverse* linked list is used because the head pointer is dynamically updated at construction time to receive the address of the latest linked list node written out at each pixel location. Once construction is complete, the head pointer value effectively contains the address of the *last* node written out, with this last node sequentially pointing to the nodes previously stored for the same pixel location.

2.4.2 Nodes Buffer

The nodes buffer stores the nodes of *all* per-pixel linked lists. We cannot allocate individual nodes buffers for every linked list since the render viewport dimensions guarantee that a huge number of them will be required (one for each pixel in the render viewport). Therefore the nodes buffer must be allocated with enough memory to accommodate all possible translucent fragments in the scene. It is the responsibility of the programmer to define this upper limit. A good heuristic to use is to base the allocation size on the render viewport dimensions multiplied by the average translucent overdraw expected. Certainly the size of the nodes buffer is likely to be very large, and may place an unreasonable burden on the video memory requirements of the OIT technique. Section 2.9 introduces a significant optimization to dramatically reduce the memory requirements of this algorithm.

The UAV for the nodes buffer is created with a built-in hardware counter initialized to zero as a way to keep track of how many fragments have been stored in the buffer.

2.4.3 Fragment Data

Each linked list node stores fragment data as well as the address of the next node. The address of the next node is stored as a `uint` while the type and size of the fragment data depends on the needs of the application. Typically the fragment structure includes fragment depth and color but other variables may be stored as needed (e.g., normal, blending mode id, etc.). Fragment depth is an essential component of the fragment structure since it will be required at linked list traversal time to correctly sort fragments prior to rendering.

It is important to point out that any additional structure member will increase the total size of the nodes buffer and therefore the total memory requirement of the OIT solution. It is therefore desirable to economize the size of the fragment structure. The implementation presented in this chapter packs the fragment color into a single `uint` type and uses the following node data structure for a total of 12 bytes per fragment:

```
struct NodeData_STRUCT
{
    uint uColor; // Fragment color packed as RGBA
    uint uDepth; // Fragment depth
    uint uNext;  // Address of next linked list node
};
```

2.5 Per-Pixel Linked List Creation

The algorithm presented does not use `DirectCompute`; instead, the storing of translucent fragments into linked lists is done via a pixel shader writing to the head pointer and nodes buffers UAVs. Earlier shader stages in the pipeline are enabled (vertex shader, but also hull shader, domain shader and geometry shader if needed) in order to turn incoming triangles into fragments that can be stored into per-pixel linked lists.

No color render target is bound at this stage although a depth buffer is still required to ensure that only translucent fragments that pass the depth/stencil test are stored in the per-pixel linked lists. Binding a depth buffer avoids the need to store translucent fragments that would result in being completely hidden by previously-rendered opaque or alpha-tested geometry.

2.5.1 Algorithm Description

A description of the algorithm used to build per-pixel linked lists follows.

- For each frame
  - *Clear head pointer buffer to 0xFFFFFFFF* (-1). This indicates that the per-pixel linked lists are all empty to start with.

- *For (each translucent fragment)*
  - *Compute and store fragment color and depth into node structure.* Calculate the fragment color as normal using lighting, texturing etc. and store it in node data structure. Pixel depth can be directly obtained from the z member of the `SV_POSITION` input.
  - *Retrieve current counter value of nodes buffer and increment counter.* The current counter value tells us how many nodes have been written into the nodes buffer. We will use this value as the offset at which to store the new node entry.
  - *Atomically exchange head pointer for this screen location with counter value.* The previous head pointer for the current screen coordinates is retrieved and set as the "next" member of the node structure. The node structure being prepared therefore points to the previous node that was written out at this pixel location. The new head pointer receives the current value of the counter as it will represent the latest node written out.
  - *Store node structure into node buffer at offset specified by counter value.* The node structure containing fragment data and next node pointer is written out to the nodes buffer at the offset specified by the current counter value. This is the latest node to be written out to the nodes buffer for these pixel coordinates; this is why the counter offset was also set as the new head pointer in the previous step.

Figure 2.1 illustrates the contents of the head pointer and nodes buffers after three translucent triangles go through the per-pixel linked list creation step.

The single pixel occupied by the orange triangle stores the current nodes buffer counter value (0) into location $[1, 1]$ in the Head Pointer Buffer and sets the previous Head Pointer value (-1) as the "next" node pointer in the node structure before writing it to the nodes buffer at offset 0.

The two pixels occupied by the green triangle are processed sequentially; they store the current nodes buffer counter values (1 and 2) into locations $[3, 4]$ and $[4, 4]$ in the head pointer buffer and set the previous head pointer values $(-1$ and $-1)$ as the "next" node pointers in the two node structures before writing them to the nodes buffer at offset 1 and 2.

The left-most pixel of the yellow triangle is due to be rendered at the same location as the orange fragment already stored. The current counter value (3) replaces the previous value (0) in the head pointer buffer at location $[1, 1]$ and the previous value is now set as the "next" node pointer in the fragment node before writing it to offset 3 in the nodes buffer.

The second pixel of the yellow triangle stores the current counter value (4) into location $[2, 1]$ and sets the previous value (-1) as the "next" node pointer before writing the node to the nodes buffer at offset 4.

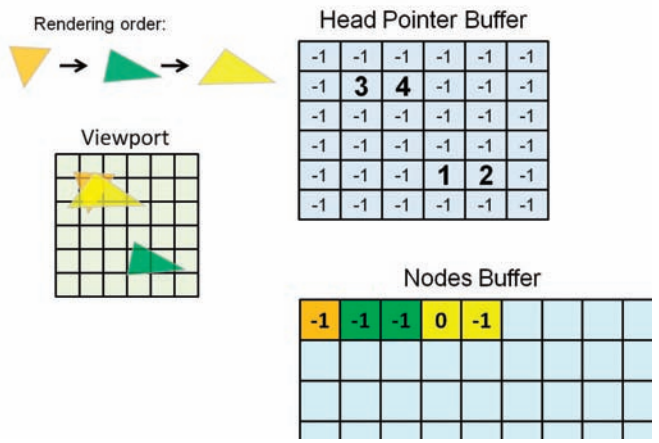


Figure 2.1. Head pointer and nodes buffers contents after rendering five translucent pixels (3 triangles) for a 6×6 render target viewport.

2.5.2 Pixel Shader Code

The pixel shader code for creating per-pixel linked lists can be found in Listing 2.1.

```
// Pixel shader input structure
struct PS_INPUT
{
    float3 vNormal : NORMAL;    // Pixel normal
    float2 vTex : TEXCOORD;     // Texture coordinates
    float4 vPos : SV_POSITION; // Screen coordinates
};

// Node data structure
struct NodeData_STRUCT
{
    uint uColor; // Fragment color packed as RGBA
    uint uDepth; // Fragment depth
    uint uNext;  // Address of next linked list node
};

// UAV declarations
RWByteAddressBuffer HeadPointerBuffer : register(u1);
RWStructuredBuffer<NodeData_STRUCT> NodesBuffer : register(u2);

// Pixel shader for writing per-pixel linked lists
[earlydepthstencil]
float PS_StoreFragments(PS_INPUT input) : SV_Target
{
    NodeData_STRUCT Node;
```

```
    // Calculate fragment color from pixel input data
    Node.uColor = PackFloat4IntoUint(ComputeFragmentColor(input));

    // Store pixel depth in packed format
    Node.uDepth = PackDepthIntoUint( input.vPos.z );

    // Retrieve current pixel count and increase counter
    uint uPixelCount = NodesBuffer.IncrementCounter();

    // Convert pixel 2D coordinates to byte linear address
    uint2 vScreenPos = uint(input.vPos.xy);
    uint uLinearAddressInBytes = 4 * ( vScreenPos.y*RENDERWIDTH +
                                       vScreenPos.x );

    // Exchange offsets in Head Pointer buffer
    // Node.uNext will receive the previous head pointer
    HeadPointerBuffer.InterlockedExchange(
        uLinearAddressInBytes, uPixelCount, Node.uNext);

    // Add new fragment entry in Nodes Buffer
    NodesBuffer[uPixelCount] = Node;

    // No RT bound so this will have no effect
    return float4(0,0,0,0);
}
```

Listing 2.1. Pixel shader for creating per-pixel linked lists.

2.6 Per-Pixel Linked Lists Traversal

Once the head pointer and nodes buffers have been filled with data, the second phase of the algorithm can proceed: traversing the per-pixel linked lists and rendering translucent pixels in the correct order.

The traversal code needs to be executed once per pixel; we therefore render a fullscreen quad (or triangle) covering the render viewport with a pixel shader that will parse all stored fragments for every pixel location. Each fragment from the corresponding per-pixel linked list will be fetched and then sorted with other fragments before a manual back-to-front blending operation will take place, starting with the color from the current render target.

2.6.1 Pixel Shader Inputs and Outputs

The head pointer and nodes buffers are now bound as pixel shader inputs, since the traversal step will exclusively be *reading* from them. We will also need a copy of the render target onto which opaque and alpha-tested geometry have previously

been rendered. This copy is needed to start the manual blending operation in the pixel shader.

The current render target viewport (onto which opaque and alpha-tested geometry has previously been rendered) is set as output, and the depth buffer that was used to render previous geometry is bound with Z-Writes disabled (the use of the depth buffer for the traversal step is explained in the Optimizations section).

2.6.2 Algorithm Description

A description of the algorithm used to parse per-pixel linked lists and render from them follows.

- For (each frame)
  - *Copy current render target contents into a background texture.* This copy is required to perform the manual blending process.
  - *For (each pixel)*
    - *Fetch head pointer for corresponding pixel coordinates and set it as current node pointer.* The screen coordinates of the current pixel are converted to a linear address used to fetch the head pointer from the head pointer buffer. The head pointer is therefore the first node address pointing to the list of all translucent fragments occupying this pixel location.
    - *While (end of linked list not reached)*
      - *Fetch current fragment from nodes buffer using current node pointer.* The current node pointer indicates where the next fragment is stored in the nodes buffer for the current pixel location.
      - *Sort fragment with previously collected fragments in front-to-back order.* The current fragment is inserted into the last position of a temporary array. The fragment is then sorted on-the-fly with previous fragments stored in the array.
      - *Set current node pointer to "next" node pointer from current node.* The current fragment has been processed; we therefore proceed to the next fragment in the list by setting the current node pointer to the "next" pointer of the current node. Fragment parsing will stop once the "next" pointer equals 0xFFFFFFFF, as this value indicates the end of the list.
    - *The background color is fetched from the background texture and set as the current color.* Back-to-front blending requires the furthermost color value to be processed first, hence the need to obtain the current value of the render target to start the blending process.

* *For (each fragment fetched from sorted list in reverse order)*
  - Perform manual back-to-front blending of current color with fragment color. The latest entry in the front-to-back sorted array is fetched, unpacked from a uint to a float4, and blended manually with the current color.
* *The final blended color is returned from the pixel shader.* This color is the correctly-ordered translucent contribution for this pixel.

2.6.3 Sorting Fragments

The algorithm used to sort fragments for correct ordering can be arbitrary because it typically has no dependencies with the OIT algorithm described in this chapter.

The implementation presented here declares a temporary array of `uint2` format containing fragment color and depth. At parsing time each fragment extracted from the per-pixel linked list is directly stored as the last entry in the array, and sorted with previous array elements using a standard insertion-sort algorithm. This approach requires an array of a fixed size, which unfortunately limits the maximum number of fragments that can be processed with this technique. The alternative option of parsing and sorting the linked list "in place" resulted in much lower performance due to repeated accesses to memory, and was also subject to limitations of the DirectX Shader Compiler regarding loop terminating conditions being dependent on the result of a UAV fetch. For those reasons, sorting via a temporary array was chosen. In general the smaller the array the better the performance since temporary arrays consume precious general purpose registers (GPRs) that affect the performance of the shader. The programmer should therefore declare the array size as an estimate of the maximum translucent overdraw that can be expected in the scene.

2.6.4 Blending Fragments

The blending process takes place once all fragments for the current pixel coordinates have been parsed and sorted in the temporary array. The blending operation used in this implementation uses the same `SRCALPHA-INVSRCALPHA` blend mode for all translucent fragments. The blending process starts with the background color and then iteratively blends the current color with each fragment color in a back-to-front order. Because the blending is performed "manually" in the pixel shader, actual hardware color/alpha blending is disabled.

A different approach that would avoid the need for copying the render target contents to a texture prior to rendering translucent fragments would be to use underblending [Bavoil 08]. Underblending allows fragments to be blended in a front-to-back order, hence avoiding the need to access the background color as a texture (the background color will be blended with the result of the manually

underblended result via actual hardware blending). However, this method imposes restrictions on the variety of per-fragment blend modes that can be used, and did not noticeably affect performance.

It is quite straightforward to modify the algorithm so that per-fragment blend modes are specified instead of adopting a blend mode common to all fragments. This modification allows translucent geometry of different types (particles, windows, smoke etc.) to be stored and processed together. In this case a bit field containing the blend mode id of each fragment is stored in the node structure (along with pixel color and depth) in the per-pixel linked list creation step. Only a few bits are required (this depends on how many different blend modes are specified— typically this shouldn't be more than a handful) and therefore the bit field could be appended to the existing color or depth member of the node structure by modifying the packing function accordingly. When the per-pixel linked lists are parsed for rendering, the blending part of the algorithm is modified so that a different code path (ideally based on pure arithmetic instructions to avoid the need for actual code divergence) is executed based on the fragment's blend mode id.

2.6.5 Pixel Shader Code

The pixel shader code for linked list traversal and rendering is given in in Listing 2.2.

```
// Pixel shader input structure for fullscreen quad rendering
struct PS_SIMPLE_INPUT
 {
    float2 vTex : TEXCOORD;     // Texture coordinates
    float4 vPos : SV_POSITION; // Screen coordinates
  };

// Fragment sorting array
#define MAX_SORTED_FRAGMENTS 18
static uint2 SortedFragments[MAX_SORTED_FRAGMENTS+1];

// SRV declarations
Buffer<uint> HeadPointerBufferSRV : register(t0);
StructuredBuffer<NodeData_STRUCT> NodesBufferSRV : register(t1);
Texture2D BackgroundTexture : register(t3);

// Pixel shader for parsing per-pixel linked lists
float4 PS_RenderFragments( PS_SIMPLE_INPUT input) : SV_Target
{

    // Convert pixel 2D coordinates to linear address
    uint2 vScreenPos = uint(input.vPos.xy);
    uint uLinearAddress = vScreenPos.y*RENDERWIDTH + vScreenPos.x;
```

```
// Fetch offset of first fragment for current pixel
uint uOffset = HeadPointerBufferSRV[uLinearAddress];

// Loop through each node stored for this pixel location
int nNumFragments = 0;
while (uOffset != 0xFFFFFFFF)
{

    // Retrieve fragment at current offset
    NodeData_STRUCT Node = NodesBufferSRV[uOffset];

    // Copy fragment color and depth into sorting array
    SortedFragments[nNumFragments] =
    uint2(Node.uColor, Node.uDepth);

    // Sort fragments front to back using insertion sorting
    int j = nNumFragments;
    [loop]while ( (j>0) &&
        (SortedFragments[max(j-1, 0)].y >
        SortedFragments[j].y) )
    {

        // Swap required
        int jminusone = max(j-1, 0);
        uint2 Tmp = SortedFragments[j];
        SortedFragments[j] = SortedFragments[jminusone];
        SortedFragments[jminusone] = Tmp;
        j--;
    }

    // Increase number of fragment if under the limit
    nNumFragments = min(nNumFragments+1,
    MAX_SORTED_FRAGMENTS);

    // Retrieve next offset
    uOffset = Element.uNext;
}

// Retrieve current color from background color
float4 vCurrentColor =
BackgroundTexture.Load(int3(input.vPos.xy, 0));

// Render sorted fragments using SRCALPHA-INVSRCALPHA
// blending
for (int k=nNumFragments-1; k>=0; k--)
{

    float4 vColor = UnpackUintIntoFloat4
    (SortedFragments[k].x);
    vCurrentColor.xyz = lerp(vCurrentColor.xyz, vColor.xyz,
                             vColor.w);
}
```

```
    // Return manually-blended color
    return vCurrentColor;
}
```

Listing 2.2. Pixel shader for parsing per-pixel linked lists.

2.7 Multisampling Antialiasing Support

Multisampling antialiasing (MSAA) is supported by the OIT algorithm presented in this chapter via a couple of minor modifications to the technique. MSAA works by performing the depth test at multiple pixel sample locations and outputting the result of the pixel shader stage (evaluated at centroid location) to all samples that passed the depth test.

2.7.1 Per-pixel Linked Lists Creation

Directly storing translucent samples into per-pixel linked lists would rapidly become prohibitive from both a memory and performance perspective due to the sheer amount of samples to store. Instead the algorithm adapted to MSAA can simply store fragment data into per-pixel linked nodes as usual, but including *sample coverage* data in the node structure.

Sample coverage is an input provided by the pixel shader stage that specifies whether samples are covered by the input primitive. Sample coverage is a bit field containing as many useful bits as the number of samples, and is passed down to the pixel shader stage via the DirectX 11-specific `SV_COVERAGE` input (Figure 2.2 illustrates the concept of sample coverage on a single pixel.):

```
// Pixel shader input structure
struct PS_INPUT
{
    float3 vNormal : NORMAL;        // Pixel normal
    float2 vTex : TEXCOORD;         // Texture coordinates
    float4 vPos : SV_POSITION;      // Screen coordinates
    uint uCoverage : SV_COVERAGE; // Pixel coverage
};
```

Only a few bits are required for sample coverage; we therefore pack it onto the depth member of the node structure using a 24:8 bit arrangement (24 bits for depth, 8 bits for sample coverage). This avoids the need for extra storage and leaves enough precision for encoding depth. The node structure thus becomes:

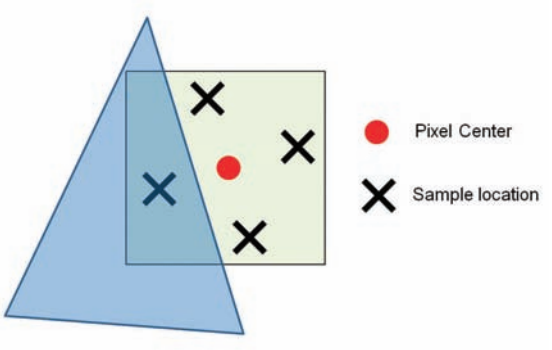


Figure 2.2. Sample coverage example on a single pixel. The blue triangle covers the third sample in a standard MSAA 4x arrangement. The input coverage to the pixel shader will therefore be equal to 0x04 (0100 in binary).

```
struct NodeData_STRUCT
{
    uint uColor;              // Fragment color packed as RGBA
    uint uDepthAndCoverage;   // Fragment depth and coverage
    uint uNext;               // Address of next linked list node
};
```

The pixel shader to create per-pixel linked lists is modified so that depth *and* coverage are now packed together and stored in the node structure:

```
// Store pixel depth and coverage in packed format
Node.uDepthAndCoverage = PackDepthAndCoverageIntoUint(
    input.vPos.z, input.uCoverage );
```

2.7.2 Per-pixel Linked Lists Traversal

The traversal of per-pixel linked lists needs to occur at *sample frequency* when outputting into a multisampled render target. This is because fragments stored in the per-pixel linked lists may affect only some samples at a given pixel location. Therefore both fragment sorting and blending must be performed per sample to ensure that translucent geometry gets correctly antialiased.

As in the non-multisampled case, a fullscreen quad is rendered to ensure that every pixel location in the destination render target is covered; one notable difference is that the pixel shader input for this pass now declares the `SV_SAMPLEINDEX` system value in order to force the pixel shader to execute at sample frequency. This value specifies the index of the sample at which the pixel shader currently executes.

When parsing per-pixel linked lists for rendering, the sample coverage in the current node is compared with the index of the sample being shaded: if the index is included in the sample coverage, then this pixel node contributes to the current sample and is therefore copied to the temporary array for sorting and later blending.

One further modification to the blending portion of the code is that the background texture representing the scene data prior to any translucent contribution is multisampled, thus the declaration and the fetch instruction are modified accordingly.

After the fullscreen quad is rendered, the multisampled render target will contain the sample-accurate translucent contribution to the scene.

2.7.3 Pixel Shader Code

The pixel shader code for parsing and rendering from per-pixel linked lists with MSAA enabled can be found in Listing 2.3. Code specific to MSAA is highlighted. Note that the shader can easily cater to both MSAA and non-MSAA cases by the use of a few well-placed `#define` statements.

```
// Pixel shader input structure for fullscreen quad rendering
// with MSAA enabled
struct PS_SIMPLE_INPUT
{
    float2 vTex : TEXCOORD;       // Texture coordinates
    float4 vPos : SV_POSITION;    // Screen coordinates
    uint uSample : SV_SAMPLEINDEX;    // Sample index
   };

// Fragment sorting array
#define MAX_SORTED_FRAGMENTS 18
static uint2 SortedFragments[MAX_SORTED_FRAGMENTS+1];

// SRV declarations
Buffer<uint> HeadPointerBufferSRV : register(t0);
StructuredBuffer<NodeData_STRUCT> NodesBufferSRV : register(t1);
Texture2DMS <float4, NUM_SAMPLES BackgroundTexture : register(t3);

// Pixel shader for parsing per-pixel linked lists with MSAA
float4 PS_RenderFragments( PS_SIMPLE_INPUT input) : SV_Target
{

    // Convert pixel 2D coordinates to linear address
    uint2 vScreenPos = uint(input.vPos.xy);
    uint uLinearAddress = vScreenPos.y*RENDERWIDTH
                          + vScreenPos.x;

    // Fetch offset of first fragment for current pixel
    uint uOffset = HeadPointerBufferSRV[uLinearAddress];
```

```
// Loop through each node stored for this pixel location
int nNumFragments = 0;
while (uOffset != 0xFFFFFFFF)
{

    // Retrieve fragment at current offset
    NodeData_STRUCT Node = NodesBufferSRV[uOffset];
    // Only include fragment in sorted list if coverage mask
    // includes the sample currently being rendered}
    uintuCoverage =
            UnpackCoverageIntoUint(Node.uDepthAndCoverage);
   if ( uCoverage & (1<<input.uSample) )}
   {
        // Copy fragment color and depth into sorting array
        SortedFragments[nNumFragments] =
            uint2(Node.uColor, Node.uDepth);

        // Sort fragments front to back using
        // insertion sorting
        int j = nNumFragments;
        [loop]while ( (j>0) &&
            (SortedFragments[max(j-1, 0)].y >
                SortedFragments[j].y) )
        {
            // Swap required
            int jminusone = max(j-1, 0);
            uint2 Tmp = SortedFragments[j];
            SortedFragments[j] = SortedFragments[jminusone];
            SortedFragments[jminusone] = Tmp;
            j--;
        }

        // Increase number of fragment if under the limit
        nNumFragments=min(nNumFragments+1,
                          MAX_SORTED_FRAGMENTS);
    }

    // Retrieve next offset
    uOffset = Element.uNext;
}

// Retrieve current sample color from background texture
float4 vCurrentColor =
 BackgroundTexture.Load(int3(input.vPos.xy, 0),
                        input.uSample);}

 // Render sorted fragments using SRCALPHA-INVSRCALPHA
 // blending
 for (int k=nNumFragments-1; k>=0; k--)
 {

    float4 vColor =
            UnpackUintIntoFloat4(SortedFragments[k].x);
```

```
            vCurrentColor.xyz = lerp(vCurrentColor.xyz, vColor.xyz,
                                     vColor.w);
        }

        // Return manually-blended color
        return vCurrentColor;
    }
```

Listing 2.3. Pixel Shader for parsing per-pixel linked lists when MSAA is enabled

2.8 Optimizations

2.8.1 Node Structure Packing

As previously mentioned in this chapter, the size of the node structure has a direct impact on the amount of memory declared for the nodes buffer. Incidentally, the smaller the size of the node structure, the better the performance, since fewer memory accesses will be performed. It therefore pays to aggressively pack data inside the node structure, even if it adds to the cost of packing/unpacking instructions in the shaders used.

The default node structure presented in previous paragraphs is three `uint` in size whereby one `uint` is used for packed RGBA color, one `uint` is used for depth and coverage, and the last `uint` is used for the next pointer. Some circumstances may allow further reduction of the structure for a performance/memory benefit; for instance, color and depth could be packed into a single `uint` (e.g., by encoding RGB color as 565 and depth as a 16-bit value (such a reduction in depth precision may need some scaling and biasing to avoid precision issues)). The "next" pointer could be encoded with 24 bits, leaving 8 bits for a combination of sample coverage and/or blend id. Such a scheme would reduce the node structure size to two `uint` (8 bytes), which is a desirable goal if the scene circumstances allow it.

2.8.2 Early Stencil Rejection in Parsing Phase

The second pass of the OIT algorithm can be accelerated by ensuring that only screen locations that have received at least one translucent fragment entry are processed. This would avoid the need for "empty" parsing of the per-pixel linked lists, improving performance in the process.

To achieve this goal, the linked lists creation phase is set up so that the stencil buffer is incremented for each fragment that passes the depth test. Once this phase is completed, the stencil buffer will contain translucent overdraw for each pixel in the scene, leaving the stencil clear value at 0 for pixel locations that haven't received any translucent fragment (i.e., for which the head pointer buffer is still 0xFFFFFFFF).

When per-pixel linked lists are parsed for rendering the stencil buffer is set up to pass if the stencil value is above 0. Early stencil rejection ensures that only pixel locations that have been touched by translucent fragments will be processed, saving on performance in the process.

2.8.3 MSAA: Fragment Resolve during Parsing

Executing the pixel shader at sample frequency is a significant performance cost compared to per-pixel execution. It is possible to replace per-sample execution for per-pixel execution for the fragment parsing and rendering pass in MSAA mode if fragments are "resolved" at the same time they are processed. Resolving fragments is a term that refers to the process of converting pixel samples into a single pixel color value that gets written onto a single-sample render target. The most common resolve function is a straight average of all samples belonging to a pixel but some variations exist (e.g., HDR-correct resolves).

To resolve fragments in the OIT rendering phase a *non-multisampled* render target has to be bound as an output. Doing so will prevent any further rendering operation requiring access to multisampled data, so it is important that this optimization is considered only if such access is no longer required. Should this condition be fulfilled, the performance improvements enabled by fragment resolving can be dramatic (up to a 50% increase was observed on a Radeon 5870 at 1024×768 with 4x MSAA) so this is certainly an optimization to consider. Because the render target bound is no longer multisampled the depth stencil buffer that was bound when storing fragments can no longer be used for early stencil rejection in the fragment parsing phase. Still, the performance boost obtained via fragment resolving outweighs the benefit of this previous optimization.

To restore per-pixel shader execution, the pixel shader input structure no longer declares `SV_SAMPLEINDEX`. Only the blending section of the per-pixel linked list parsing shader needs further code modification to enable fragment resolving. After fragments have been fetched and sorted in the temporary array, the algorithm needs to determine the blending contribution for each sample in the destination pixel. Hence the next node picked from the sorted list will add its blending contribution to a sample only if its pixel coverage includes the sample currently being processed. Once the blending contribution for all samples has been determined, the final average (resolve) operation takes place and the resulting color is output to the destination render target.

The blending portion of the per-pixel linked list parsing shader that implements fragment resolve in MSAA mode can be found in Listing 2.4.

```
// Retrieve color of each sample in the background texture
float3 vCurrentColorSample[NUM_SAMPLES];
[unroll]for (uint uSample=0; uSample<NUM_SAMPLES; uSample++)
{
```

```
        vCurrentColorSample[uSample] =
            BackgroundTexture.Load(int3(input.vPos.xy, 0), uSample);
    }

    // Render fragments using SRCALPHA-INVSRCALPHA blending
    for (int k=nNumFragments-1; k>=0; k--)
    {
        // Retrieve fragment color
        float4 vFragmentColor=
                    UnpackUintIntoFloat4(SortedFragments[k].x);

        // Retrieve sample coverage
        uint uCoverage =
                    UnpackCoverageIntoUint(SortedFragments[k].y);

        // Loop through each sample
        [unroll]for (uint uSample=0; uSample<NUM_SAMPLES; uSample++)
        {
            // Determine if sample is covered by sample coverage
            float fIsSampleCovered= (uCoverage & (1<<uSample)) ?
                                        1.0 : 0.0;

            // Blend current color sample with fragment color
            // if covered. If the sample is not covered the color
            // will be unchanged
            vCurrentColorSample[uSample].xyz = lerp(
                    vCurrentColorSample[uSample].xyz,
                    vFragmentColor.xyz,
                    vFragmentColor.w * fIsSampleCovered);
        }
    }

    // Resolve samples into a single color
    float4 vCurrentColor = float4(0,0,0,1);
    [unroll]for (uint uSample=0; uSample<NUM_SAMPLES; uSample++)
    {
        vCurrentColor.xyz += vCurrentColorSample[uSample];
    }
    vCurrentColor.xyz *= (1.0/NUM_SAMPLES);

    // Return manually-blended color
    return vCurrentColor;
```

Listing 2.4. Blending and resolving fragments

2.9 Tiling

2.9.1 Tiling as a Memory Optimization

Tiling is a pure memory optimization that considerably reduces the amount of video memory required for the nodes buffer (and to a lesser extent the head

Figure 2.3. Opaque contents of render target prior to any OIT contribution.

pointer buffer). Without tiling, the memory occupied by both buffers can rapidly become huge when fullscreen render target resolutions are used. As an example a standard HD resolution of 1280×720 with an estimated average translucent overdraw of eight would occupy a total of $1280 \times 720 \times 8 \times \text{sizeof}(\text{node structure size})$ bytes for the nodes buffer only, which equates to more than 168 megabytes with a standard node structure containing 3 units (color, depth and next pointer).

Instead of allocating buffers for the full-size render target, a single, smaller rectangular region (the "tile") is used. This tile represents the extent of the

Figure 2.4. Translucent contribution to the scene is added to the render target via regular tiling. Each rectangular area stores fragments in the tile-sized head pointer and nodes buffers and then parses those buffers to add correctly ordered translucency information to the same rectangle area. In this example the tile size is 1/15 of the render target size, and a total of 15 rectangles are processed.

area being processed for OIT in a given pass. Since the tile is typically smaller than the render size, this means multiple passes are needed to calculate the full-size translucent contributions to the scene. The creation of per-pixel linked lists is therefore performed on a per-tile basis, after which the traversal phase fetches nodes from the tile-sized head pointer and nodes buffers to finally output the resulting color values onto the rectangular region corresponding to the tile being processed in the destination render target. As an example, a standard HD resolution of 1280×720 would take 15 passes with a tile size of 256×240 for the screen to be fully covered. Figure 2.3 shows a scene with translucent contributions yet to be factored in. Figure 2.4 shows the same scene with translucency rendered on top using a set of tile-sized rectangular regions covering the whole render area.

2.9.2 Transformation Overhead

Because multiple passes are required, any objects crossing tile boundaries has to be transformed more than once (once for each tile-sized screen region they contribute to), which can impact performance when objects with higher transformation costs (complex vertex skinning, advanced tessellation algorithms, use of long geometry shaders, etc.) are employed. Ideally, translucent geometry should be decomposed into subobjects to minimize transformation overlap between tile regions. Smaller objects such as particles will particularly benefit from the tiling approach, since overlap between tile regions is likely to be minimal (unless the tiles themselves are small), and boundaries between particles belonging to different rectangle regions are well defined.

2.9.3 Tile Size

Tile size can be arbitrary; it may be a multiple of the render target dimensions, but this is not a strict requirement. Typically tile size will be dictated by the amount of free video memory available for OIT, since larger dimensions will lead once again to significant amounts of memory being consumed. There is also a direct correlation between tile size and performance, since the smaller the tile size, the higher the number of passes required to cover the render area. Thus, it is generally better to allocate a larger tile size if memory can be spared for this purpose.

2.9.4 Minimum Rectangle Optimization

It is not necessary to cover the *full* render target area with multipass tiling. Instead one needs to cover only the *minimum* rectangle area of the render target that will actually receive translucent contributions. This minimum rectangle area can be determined by using bounding volumes transformed to screen space in order to retrieve the 2D screen coordinates extents of all translucent geometry. Once the bounding volumes of all translucent meshes have been transformed,

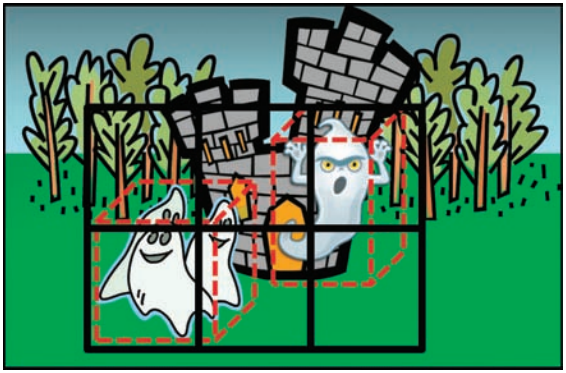


Figure 2.5. Translucent contribution to the scene is added to the render target via optimized tiling. Only tiles enclosing the bounding geometry of the translucent characters are processed. The bounding boxes of translucent geometry are transformed to screen space and the combined extents of the resulting coordinates define the minimum rectangle area that will be processed. This minimum rectangle area is covered by as many tile-sized rectangular regions as required (six in this example). Each of those regions performs fragment storing and rendering using a single pair of tile-sized head pointers and nodes buffers.

the minimum and maximum dimensions (in X and Y) of the combined set will define the rectangle area of translucent contributions. The minimum rectangle optimization typically allows a reduction in the number of tiles to process when parsing and rendering fragments from linked lists. In order to render a minimum number of tiles, it is desirable to ensure that the bounding geometry used is as tight as possible; for example, axis-aligned bounding boxes are likely to be less effective than arbitrary-aligned bounding boxes or a set of bounding volumes with a close fit to the meshes involved.

Because this optimization covers only a portion of the screen, the previous contents of the render target will need to be copied to the destination render target, at least for those regions that do not include translucent contribution. This copy can be a full-size copy performed before the OIT step, or stencil-based marking can be used to transfer only the rectangle regions that did not contain any translucency.

Figure 2.5 illustrates the optimized covering of tiles to cover only the 2D extents of translucent contributions to the scene.

2.10 Conclusion

The OIT algorithm presented in this chapter allows significant performance savings compared to other existing techniques. The technique is also robust, allowing

different types of translucent materials to be used as well as supporting hardware multisampling antialiasing. Although video memory requirements can become unreasonable when larger viewports are employed, the use of tile optimizations allows a trade-off between memory and performance while still retaining the inherent advantages of this technique. The use of per-pixel linked lists can be adapted to techniques other than order-independent transparency, because they can store a variety of per-pixel data for arbitrary purposes: see Part III, Chapter 3 in this volume for an example.

2.11 Acknowledgments

I would like to thank Holger Gruen and Jakub Klarowicz for coming up with the original concept of creating per-pixel linked lists in a DirectX 11-class GPU environment.

Bibliography

[Bavoil 08] Louis Bavoil and Kevin Myers. "Order-Independent Transparency with Dual Depth Peeling." White paper available online at http://developer.download.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf, 2008.

[Carpenter 84] L. Carpenter. "The A-Buffer, An Antialiased Hidden Surface Method." *Computer Graphics (SIGGRAPH)* 18:3 (1984), 103–108.

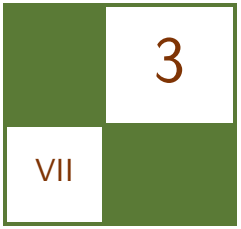
[Everitt 01] Cass Everitt. "Interactive Order-Independent Transparency." White paper available online at http://developer.nvidia.com/object/Interactive_Order_Transparency.html, 2001.

[Myers 07] Kevin Myers and Louis Bavoil. "Stencil Routed A-Buffer." *SIGGRAPH '07: ACM SIGGRAPH 2007 Sketches*, Article 21. New York: ACM, 2007.

[Thibieroz 08] Nicolas Thibieroz. "Robust Order-Independent Transparency via Reverse Depth Peeling," In *ShaderX*^{6}, edited by Wolfgang Engel, pp. 211–226. Hingham, MA; Charles River Media, 2008.

[Thibieroz 10] Nicolas Thibieroz and Holger Gruen. "OIT and Indirect Illumination using DX11 Linked Lists," *GDC 2010 Presentation from the Advanced D3D Day Tutorial.* Available online at http://developer.amd.com/documentation/presentations/Pages/default.aspx, 2010

[Yang 10] Jason Yang, Justin Hensley, Holger Gruen, and Nicolas Thibieroz. "Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum, Eurographics Symposium on Rendering 2010* 29:4 (2010), 1297–1304.



Simple and Fast Fluids

Martin Guay, Fabrice Colin, and Richard Egli

3.1 Introduction

In this chapter, we present a simple and efficient algorithm for the simulation of fluid flow directly on the GPU using a single pixel shader. By temporarily relaxing the incompressibility condition, we are able to solve the full Navier-Stokes equations over the domain in a single pass.

3.1.1 Simplicity and Speed

Solving the equations in an explicit finite difference scheme is extremely simple. We believe that anybody who can write the code for a blur post-process can implement this algorithm and simulate fluid flow efficiently on the GPU. The code holds is less then 40 lines and is so simple we actually had the solver running in FxComposer (see FxComposer demo). Solving every fluid cell (texel) locally in a single pass is not only simple, but also quite fast. In fact, the implementation of

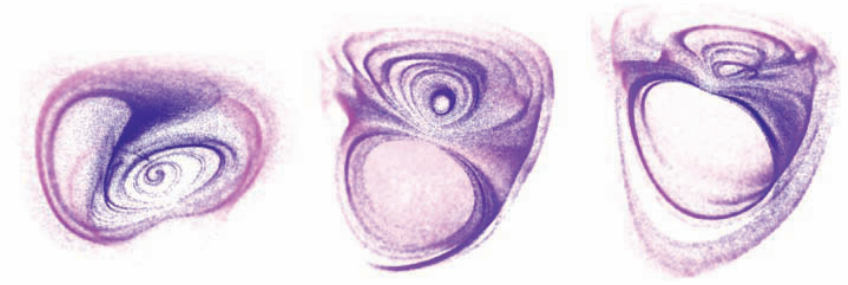


Figure 3.1. 3D simulation with 100k particles visualization.

this algorithm on the GPU is at least 100 times faster than on the CPU.[1] In this chapter, we show how to couple the two equations of the classical Navier-Stokes equations into a single-phase process; a detailed explanation of the algorithm along with example code follows.

3.2 Fluid Modeling

The greatest feature of physics-based modeling in computer graphics is the ability of a model to cope with its environment and produce realistic motion and behavior. Attempting to animate fluids nonphysically is, in our experience, a nontrivial task. In order to physically model the motion of fluids, the simplified classical Navier-Stokes equations for incompressible fluids are a good description of such mechanics, and a solver based on this model is capable of simulating a large class of fluid-like phenomena. Fortunately a deep understanding of the partial differential equations involved is not required in order to implement such a solver.

3.2.1 Navier-Stokes Equations

The Navier-Stokes equations, widely used in numerous areas of fluid dynamics, are derived from two very simple and intuitive principles of mechanics: the conservation of momentum (Equation (3.1)) and the conservation of mass (Equation (3.2)).

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = -\nabla P + \rho\mathbf{g} + \mu\nabla^2\mathbf{u}, \tag{3.1}$$

where $\mathbf{u}$ is a velocity vector, $\mathbf{g}$ is the gravitational acceleration vector, μ the viscosity, P the pressure and ∇^2 stands for the Laplacian operator $\partial^2/\partial x^2 + \partial^2/\partial y^2 + \partial^2/\partial z^2$.

$$\frac{\partial \rho}{\partial t} + \nabla\cdot(\rho\mathbf{u}) = 0, \tag{3.2}$$

where $\nabla\cdot$ represents the divergence operator. These equations also have to be supplemented with boundary conditions of Dirichlet, Neumann, or even of Robin type. Usually, the incompressibility condition (see Equation (3.3)) is imposed on the fluid by assuming that its density ρ remains constant over time. Using the latter assumption, Equation (3.2) simplifies to

$$\nabla\cdot\mathbf{u} = 0. \tag{3.3}$$

[1] The simulation runs on the CPU at 8.5 fps with 4 threads on an Intel Core 2 Quad at 2.66 GHz simulating only the velocity field over a 256×256 grid. Keeping the same grid size, the simulation of both velocity and density fields runs at more than 2500 fps on a Geforce 9800 GT using 32-bit floating point render targets. Note that 16-bit floating point is sufficient to represent velocities.

Now one of the main features of the formulation of the Navier-Stokes equations illustrated here, is the possibility, when working with regular grid domains, to use classical finite differences schemes.

3.2.2 Density-Invariance Algorithm

Unfortunately, a direct application of Equation (3.1) results in a nonzero divergence field $\mathbf{u}$ (i.e. (3.3) is no longer satisfied by the vector field $\mathbf{u}$). A lot of popular methods for simulating fluids consist of two main steps. First some temporary compressibility is allowed when Equation (3.1) is solved, and second, a correction is applied to the vector field obtained, in order to fulfill the incompressibility condition. This correction can be done by considering a projection of the resulting vector $\mathbf{w}$ field onto its divergence-free part [Stam 99]. The latter projection can also be performed in the spirit of the smoothed particle hydrodynamics (SPH) method (see for instance [Colin et al. 06]). Another way to deal with this incompressibility problem is to take advantage of the relation between the divergence of the vector field and the local density given by Equation (3.2), by trying to enforce a density invariance (see among others, [Premože et al. 03]). Recently, some techniques combining the two preceding approaches were studied (for an exhaustive study of all the previous techniques in the SPH context, see [Xu et al. 09]).

We choose an approach based on density invariance. It is interesting to note that an algorithm based on the previous approach has proven to be stable for the SPH method [Premože et al. 03]. First, observe that Equation (3.2) can be rewritten as

$$\frac{\partial \rho}{\partial t} = -\nabla \rho \cdot \mathbf{u} - \rho \nabla \cdot \mathbf{u},$$

clearly illustrating the link between the divergence of the vector field and the variation of the local density. After solving the above equation for density, a corrective pressure field could simply be given as

$$P = K(\rho^n - \rho_0), \tag{3.5}$$

where ρ_0 is the rest (initial) density and where the constant K is often chosen according to the gas-state equation (see [Desbrun and Cani 96] or [Muller et al. 03]).

The corrective field P (a density-invariant field) could be interpreted as an internal pressure whose gradient corrects the original velocity field to get a null divergence vector field. Since we are interested only in its derivative, there is no need to retain the P variable and the corresponding correction to be applied is simply given by

$$\nabla P = K \nabla \rho.$$

Now before jumping directly to code, we first need to discretize the formulation.

3.2.3 From Math to Code: Numerical Scheme

One of the main features of the formulation used in this chapter is the ability to use, along a regular grid, simple finite differences. Since the texture is the default data structure on a GPU, a regular grid holding four values per cell is a natural choice for the spatial discretization of the domain. Also, since this is an Eulerian formulation, the spatial discretization stays fixed throughout the simulation and the neighborhood of an element, the elements (in our case, the texels) around it, will always remain the same, greatly simplifying the solver's code.

The simplicity of finite differences along a one-phase coupling of both momentum and mass conservation equations through a density-invariant field enables the elaboration of a very simple algorithm to solve fluid flow in a single step. Note that other grid-based methods on the GPU exist and the interested reader can refer to [Crane et al. 07] for a multistep but unconditionally stable simulation or to [Li et al. 03] (also available in *GPU Gems 2*) for a lattice-based simulation.

3.3 Solver's Algorithm

A solution to the Navier-Stokes equations is a vector-valued function $\mathbf{u}$ and a scalar-valued function ρ which satisfies the momentum and mass conservation equations. These functions are spatially discretized on a texture where quantities $\mathbf{u}$ and ρ are stored at the texel's center. In order to update a solution $\mathbf{u}$ and ρ from time t_n to time t_{n+1}, we traverse the grid once and solve every texel in the following manner:

1. Solve the mass conservation equation for density by computing the differential operators with central finite differences and integrating the solution with the forward euler method.

2. Solve the momentum conservation equation for velocity in two conceptual steps:

   (a) Solve the transport equation using the semi-Lagrangian scheme.

   (b) Solve the rest of the momentum conservation equation using the same framework as in Step 1.

3. Impose Neumann boundary conditions.

3.3.1 Conservation of Mass

In order to compute the right-hand side of the mass conservation equation, we need to evaluate a gradient operator ∇ for a scalar-valued function ρ and a divergence operator $\nabla\cdot$ for a vector-valued function $\mathbf{u}$ both, respectively, expressed

using central finite differences as follows:

$$\nabla\rho^n_{i,j,k} = \left(\frac{\rho^n_{i+1,j,k} - \rho^n_{i-1,j,k}}{2\Delta x}, \frac{\rho^n_{i,j+1,k} - \rho^n_{i,j-1,k}}{2\Delta y}, \frac{\rho^n_{i,j,k+1} - \rho^n_{i,j,k-1}}{2\Delta z}\right),$$

$$\nabla \cdot u^n_{i,j,k} = \frac{u^n_{i+1,j,k} - u^n_{i-1,j,k}}{2\Delta x} + \frac{v^n_{i,j+1,k} - v^n_{i,j-1,k}}{2\Delta y} + \frac{w^n_{i,j,k+1} - w^n_{i,j,k-1}}{2\Delta z}.$$

And finally, an integration is performed using forward Euler over a time step Δt:

$$\rho^{n+1}_{i,j,k} = \rho^n_{i,j,k} + \Delta t(-\nabla\rho^n_{i,j,k} \cdot u^n_{i,j,k} - \rho^n_{i,j,k}\nabla \cdot u^n_{i,j,k}).$$

Indeed, there exist other finite difference schemes. For instance, one could use upwinding for the transport term or literally semi-Lagrangian advection. Unfortunately, the latter results in much numerical dissipation; an issue covered in Section 3.3.2.

3.3.2 Conservation of Momentum

We solve the momentum conservation equation for velocity $\mathbf{u}$ in two conceptual steps. The first consists of solving the transport equation $\frac{\partial \mathbf{u}}{\partial t} = -\mathbf{u} \cdot \nabla\mathbf{u}$ with a semi-Lagrangian scheme, then solving the rest of the momentum conservation equation ($\frac{\partial \mathbf{u}}{\partial t} = -\frac{\nabla P}{\rho} + \mathbf{g} + \frac{\mu}{\rho}\nabla^2\mathbf{u}$) with central finite differences and forward Euler integration.

Semi-Lagrangian scheme. First introduced to computer graphics by Jos Stam in the paper *Stable Fluids* [Stam 99], the following scheme is quite useful for solving the generic transport equation given by

$$\frac{\partial \phi}{\partial t} = -\mathbf{u} \cdot \nabla\phi, \tag{3.8}$$

at the current texel's position $\mathbf{x}_{i,j,k}$ with

$$\phi^{n+1}_{i,j,k}(\mathbf{x}_{i,j,k}) = \phi^n(\mathbf{x}_{i,j,k} - \Delta t\mathbf{u}^n_{i,j,k}). \tag{3.9}$$

The idea is to solve the transport equation from a Lagrangian viewpoint where the spatial discretization element holding quantities (e.g., a particle) moves along the flow of the fluid, and answer the following question: where was this element at the previous time step if it has been transported by a field $\mathbf{u}$ and ends up at the current texel's center at the present time? Finally, we use the sampled quantity to set it as the new value of the current texel.

Now when solving the transport equation for velocity, Equation (3.8) becomes $\frac{\partial \mathbf{u}}{\partial t} = -\mathbf{u} \cdot \nabla\mathbf{u}$ and is solved with

$$\mathbf{u}^{n+1}_{i,j,k}(\mathbf{x}_{i,j,k}) = \mathbf{u}^n(\mathbf{x}_{i,j,k} - \Delta t\mathbf{u}^n_{i,j,k}).$$

This method is not only straightforward to implement on the GPU with linear samplers, but is also unconditionally stable. Unfortunately, quantities advected in this fashion suffer from dramatic numerical diffusion and higher-order schemes exist to avoid this issue, such as McCormack schemes discussed in [Selle et al. 08]. These schemes are especially useful when advecting visual densities as mentioned in Section 3.5.

Density invariance and diffusion forces. After solving the transport term, the rest of the momentum conservation equation is solved with central finite differences. Here is a quick reminder of the part of Equation (3.1) which is not yet solved:

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{\nabla P}{\rho} + \mathbf{g} + \frac{\mu}{\rho}\nabla^2 \mathbf{u}.$$

As mentioned earlier, the gradient of the density-invariant field ∇P is equivalent to the density gradient $\nabla \rho$, i.e. $\nabla P \simeq K \nabla \rho$. Since we already computed the density gradient $\nabla \rho^n_{i,j,k}$ when solving the mass conservation equation with Equation (3.3.1), we need only to scale by K in order to compute the "pressure"" gradient ∇P. As for the diffusion term, a Laplacian ∇^2 must be computed. This operator is now expressed using a second-order central finite difference scheme:

$$\nabla^2 \mathbf{u}^n_{i,j,k} = (L(u),\, L(v),\, L(w)),$$

where

$$L(f) = \left(\frac{f^n_{i+1,j,k} - 2f^n_{i,j,k} + f^n_{i-1,j,k}}{(\Delta x)^2} + \frac{f^n_{i,j+1,k} - 2f^n_{i,j,k} + f^n_{i,j-1,k}}{(\Delta y)^2} + \frac{f^n_{i,j,k+1} - 2f^n_{i,j,k} + f^n_{i,j,k-1}}{(\Delta z)^2} \right),$$

for every twice continuously differentiable function f. Finally, an integration is performed using forward Euler over a time step Δt:

$$\mathbf{u}^{n+1}_{i,j,k} = \mathbf{u}^n_{i,j,k} + \Delta t(-S\nabla \rho^n_{i,j,k} + \mathbf{g} + \nu \nabla^{\mathbf{2}} \mathbf{u}^{\mathbf{n}}_{\mathbf{i,j,k}}).$$

Since the density ρ value should not vary much from ρ_0, we can interpret the $\frac{1}{\rho}$ scale as a constant held by $\nu := \frac{\mu}{\rho_0}$ and $S := K\frac{(\Delta x)^2}{\Delta t \rho_0}$. One can see how we also scale by $\frac{(\Delta x)^2}{\Delta t}$ which seems to give better results (we found $(\Delta x)^2$ while testing over a 2D simulation) and a sound justification has still to be found and will be the subject of future work.

Up to now we solved the equations without considering boundary conditions (obstacles) or numerical stability. These two topics will be covered in the next two sections.

3.3.3 Boundary Conditions

Boundary conditions are specified at the boundary (surface) of the fluid in order for the fluid to satisfy specific behaviors. They are essential for the interactions between the fluid and the different types of matter such as solids (obstacles) in the domain. Neumann boundary conditions are considered for fluid-solid interactions. Our method does not provide free surface boundaries for fluid-fluid interactions such as liquid and air interactions necessary for water animations. When simulating smoke or fire, for instance, it is possible to consider the air and gas as a single fluid. Therefore, only Neumann boundary conditions are required. Hence in the proposed implementation, computational cells are tagged as either fluid or boundary cells. Note velocities and densities are defined on all cell types. Before discussing in-depth Neumann boundary conditions, it is convenient to first consider the most simple boundary condition: the Dirichlet boundary condition that means "to set directly." Hence, one could set the border cells to null velocities and densities to an initial value and make sure the simulation works well before introducing obstacles.

Neumann boundary conditions. Computational cells are either tagged as fluid or boundary cells. Note velocities and densities are defined on all cell types. The treatment of obstacles requires the use of boundary conditions on the solution $\mathbf{u}$ and are usually of Neumann type. The simplest boundary condition is the Dirichlet boundary condition, which specifies the value the solution needs to take on the boundary; hence, one could set the border cells to null velocities and densities to an initial value and make sure the simulation works well before considering obstacles.

The Neumann boundary condition is a type of boundary condition that specifies the values of the derivative in the direction of the outward normal vector at the boundary. To keep the fluid from entering obstacles, this condition would result in having the obstacle's corresponding boundary cells fulfill $\frac{\partial f}{\partial n} = 0$ for every component $f \in \{u,\ v,\ w\}$ of the vector function $\mathbf{u}$; therefore, the fluid cells around a boundary cell need to be correctly adjusted in order to satisfy this condition. The derivative in the normal direction at such a boundary cell is quite

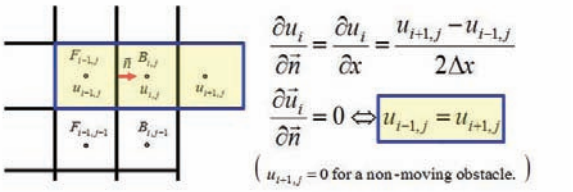


Figure 3.2. Neumann boundary conditions.

trivial when assuming that the walls of an obstacle are always coincident with the face of a computational cell (i.e., obstacles would completely fill computational cells). With this assumption, the derivative $\frac{\partial f}{\partial n}$ is either given by $\pm\frac{\partial u}{\partial x}$, $\pm\frac{\partial v}{\partial y}$, or $\pm\frac{\partial w}{\partial z}$ for u, v, and w, respectively. As an example, one can see how the fluid cell $F_{i-1,j}$ is adjusted according to the derivative of the boundary cell $B_{i,j}$ in Figure 3.2.

The true difficulty is the actual tracking of obstacles, specifically when working with dynamic 3D scenes in which objects must first be voxelized in order to be treated by the algorithm. See [Crane et al. 07] for a possible voxelization method.

3.3.4 Stability Conditions

Explicit integration is very simple from both analytical and programmatic points of view, but is only conditionally stable; meaning the time step value Δt has an upper bound defined by a ratio relative to the spatial resolution $\Delta\mathbf{x}$ over the function's range $\mathbf{u}$:

$$\Delta t < \max\left\{\left|\frac{\Delta x}{u}\right|, \left|\frac{\Delta y}{v}\right|, \left|\frac{\Delta z}{w}\right|\right\}.$$

This condition must be satisfied everywhere in the domain.

3.4 Code

Short and simple code for the 2D solver is presented in this section. In two dimensions, the x- and y-components hold velocities and the z-component holds the density. Setting $\Delta x = \Delta y = 1$ greatly simplifies the code. A 3D demo is also available on accompanying web materials ($K \simeq 0.2, \Delta t = 0.15$).

```
///< Central Finite Differences Scale.
float2 CScale = 1.0f/2.0f;

float S=K/dt;

float4 FC = tex2D(FieldSampler,UV);
float3 FR = tex2D(FieldSampler,UV+float2(Step.x,0));
float3 FL = tex2D(FieldSampler,UV-float2(Step.x,0));
float3 FT = tex2D(FieldSampler,UV+float2(0,Step.y));
float3 FD = tex2D(FieldSampler,UV-float2(0,Step.y));

float4x3 FieldMat = {FR,FL,FT,FD};

//du/dx,du/dy
float3 UdX = float3(FieldMat[0]-FieldMat[1])*CScale;
float3 UdY = float3(FieldMat[2]-FieldMat[3])*CScale;

float Udiv = UdX.x+UdY.y;
```

```
float2 DdX = float2(UdX.z,UdY.z);

///<
///< Solve for density.
///<
FC.z -=  dt*dot(float3(DdX,Udiv),FC.xyz);
///< Related to stability
FC.z = clamp(FC.z,0.5f,3.0f);

///<
///< Solve for velocity.
///<
float2 PdX = S*DdX;
float2 Laplacian = mul((float4)1,(float4x2)FieldMat)-4.0f*FC.xy;
float2 ViscosityForce = v*Laplacian;

///< Semi-Lagrangian advection.
float2 Was  = UV - dt*FC.xy*Step;
FC.xy = tex2D(FieldLinearSampler,Was).xy;

FC.xy += dt*(ViscosityForce - PdX + ExternalForces);

///< Boundary conditions.
for (int i=0; i<4; ++i)
{
if (IsBoundary(UV+Step*Directions[i]))
{
float2 SetToZero = (1-abs(Directions[i]));
FC.xy *= SetToZero;
}
}
return FC;
```

Listing 3.1. 2D solver, Shader model 2_a.

3.5 Visualization

One of the disadvantages of the Eulerian formulation is the lack of geometric information about the fluid. So far, we have captured its motion with the velocity field, but we still don't know its shape. Nevertheless, there are many ways to visualize a fluid. In this section we briefly discuss two simple techniques. The first consists of advecting particles under the computed velocity field and the second, of advecting a scalar density field.

3.5.1 Particles

Using particles in a one-way interaction with the fluid is by far the most simple and efficient technique for visualizing a fluid. Since the velocity field is computed

on the GPU, the whole system can run independently with very few interactions with the CPU. Once the particles are initialized, we sample only the velocity field in order to update their positions as illustrated in the following code:

```
v = Field.SampleLevel(LinearSampler,PosToUV(Particle.Pos),0);
Particle.Pos  += dt*v.xyz;
```

3.5.2 Smoke-Fire Density Field

It is possible to simulate smoke and fire by iteratively solving the convection-diffusion equation for a scalar density field (see Figures 3.3 and 3.4). Rendering such a scalar field in 2D is quite simple since only a texture holding the density field needs to be rendered. As for 3D fields, a volume rendering technique is required. Here are the governing equations for both smoke (Equation (3.13)) and fire (Equation (3.14)), respectively:

$$\frac{\partial \phi}{\partial t} = \mathbf{u} \cdot \nabla \phi + \mathbf{k} \nabla^{\mathbf{2}} \phi, \tag{3.13}$$

$$\frac{\partial \phi}{\partial t} = \mathbf{u} \cdot \nabla \phi + \mathbf{k} \nabla^{\mathbf{2}} \phi - \mathbf{c}, \tag{3.14}$$

where ϕ is a scalar density, k a diffusion coefficient and c a reaction constant for fire.

Numerical schemes to solve this equation are abundant, and the one which maps best to the GPU is the semi-Lagrangian method (see Equation (3.9)) but unfortunately, the solution loses much detail as dissipation occurs from this numerical scheme—which in turn enables the omission of the diffusion term from the equation. To address this problem and achieve more compelling visual results, we strongly suggest using the three-pass MacCormack method described in [Selle et al. 08]. This scheme has second-order precision both in space and time, therefore keeping the density from losing its small scale features and numerically dissipating its quantity as drastically as with the first-order semi-Lagrangian method. To add more detail to the simulation, one could also amplify the vorticity of the flow (the velocity field) with vorticity confinement, a method discussed in the context of visual smoke simulation in [Fedkiw et al. 01].

3.6 Conclusion

Many algorithms could be generated from this one-phase coupling of both equations through a density-invariant field. We hope the one illustrated here serves as a good basis for developers seeking to make use of interactive fluids in their applications.

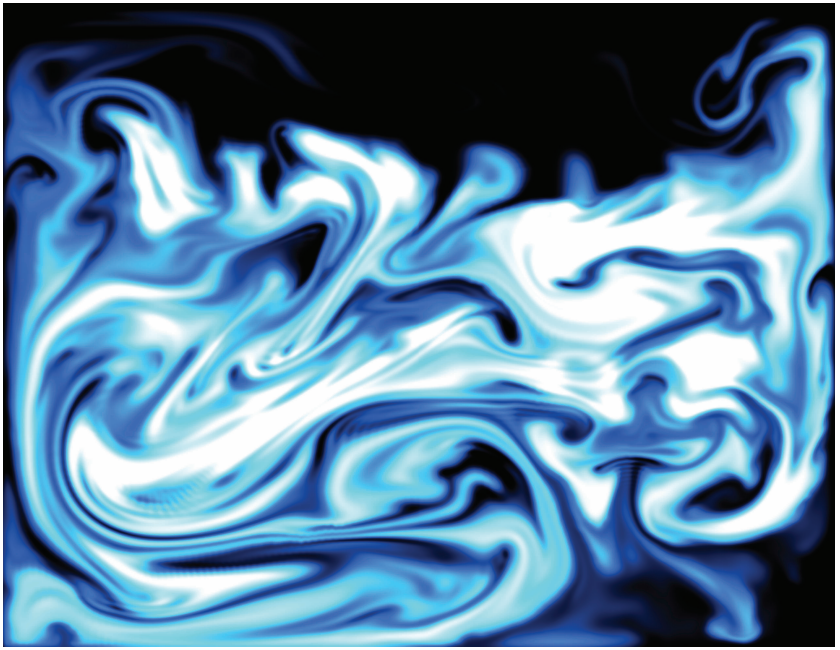


Figure 3.3. Smoke density over a 512×512 density and 256×256 fluid simulation grid.

Figure 3.4. Fire density over a 512×512 density and 256×256 fluid simulation grid.

Bibliography

[Colin et al. 06] F. Colin, R. Egli, and F.Y. Lin. "Computing a Null Divergence Velocity Field using Smoothed Particle Hydrodynamics." *Journal of Computational Physics* 217:2 (2006), 680–692.

[Crane et al. 07] K. Crane, I. Llamas, and S. Tariq. "Real-Time Simulation and Rendering of 3D Fluids." In *GPU Gems 3*, pp. 633–675. Reading, MA: Addison-Wesley, 2007.

[Desbrun and Cani 96] M. Desbrun and M.P. Cani. "Smoothed Particles: A New Paradigm for Animating Highly Deformable Bodies." In *Proceedings of EG Workshop on Animation and Simulation*, pp. 61–76. Berlin-Heidelberg: Springer-Verlag, 1996.

[Fedkiw et al. 01] R. Fedkiw, J. Stam, and H. W. Jensen. "Visual Simulation of Smoke." *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 15–22.

[Li et al. 03] W. Li, X. Wei, and A. Kaufman. "Implementing Lattice Boltzmann Computation on Graphics Hardware." *The Visual Computer* 19:7–8 (2003), 444–456.

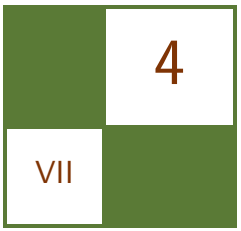
[Muller et al. 03] M. Muller, D. Charypar, and M. Gross. "Particle-Based Fluid Simulation for Interactive Applications." pp. 154–159. New York: ACM, 2003.

[Premože et al. 03] S. Premože, T. Tasdizen, J. Bigler, A. Lefohn, and R.T. Whitaker. "Particle-Based Simulation of Fluids." *Computer Graphics Forum (Proceedings of Eurographics)* 22:3 (2003), 401–410.

[Selle et al. 08] A. Selle, R. Fedkiw, K. Byungmoon, L. Yingjie, and R. Jarek. "An Unconditionally Stable MacCormack Method." *Journal of Scientific Computing* 35:2–3 (2008), 350–371.

[Stam 99] J. Stam. "Stable Fluids." In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 121–128. New York: ACM, 1999.

[Xu et al. 09] R. Xu, P. Stansby, and D. Laurence. "Accuracy and Stability in Incompressible SPH (ISPH) Based on the Projection Method and a New Approach." *Journal of Computational Physics* 228:18 (2009), 6703–6725.



A Fast Poisson Solver for OpenCL using Multigrid Methods

Sebastien Noury, Samuel Boivin, and Olivier Le Maître

4.1 Introduction

Many techniques in computer graphics are based on mathematical models to realistically simulate physical phenomena, such as fluid dynamics [Stam 99], or to deform and merge complex object meshes together [Yu et al. 04].

Many of these mathematical models involve the solution of Poisson partial differential equations, or more general elliptic equations, making the availability of efficient Poisson solvers crucial, particularly for real-time simulation. For example, the simulation of incompressible fluid flows often relies on projection-correction techniques where the pressure fields are solutions of a Poisson equation. Solving this equation is important, not only to obtain realistic flow dynamics, but also for the stability of the simulation. In fact, many efforts have been dedicated to the development of fast and stable fluid solvers [Stam 99]; the solution of the Poisson pressure equation constitutes the most time-consuming part of these algorithms.

This chapter presents an implementation of various iterative methods for the resolution of Poisson equations on heterogeneous parallel computers. Currently, most fast Poisson solvers implement the simple Jacobi method. After reviewing Jacobi iterative methods and variants in Section 4.3, we introduce more advanced iterative techniques based on multiscale iterations on a set of embedded grids, namely the so-called multigrid methods in Section 4.4. For all of these methods, we provide some theoretical background and discuss their efficiency and complexity with regard to their implementation. We particularly detail the multigrid method which involves several operators whose implementation is critical to efficiency. In Section 4.5, we provide a tutorial for the OpenCL implementation of the various algorithms, which are subsequently tested and compared in

Section 4.6. We end the chapter with a discussion of the efficiency of the methods in Section 4.7. Specifically, we show that although more complex to implement, the multigrid method allows for a significant reduction of both the number of iterations and the length of computation time compared with the simpler fixed-grid iterative methods. Hasty developers can skip directly to the implementation in (Section 4.5) and refer later to the theoretical background in Section 4.2.

4.2 Poisson Equation and Finite Volume Method

In this section, we introduce the Poisson equation, the resolution of which is the focus of the present chapter. We then describe the finite volume discretization of the Poisson equation and, finally, discuss the boundary conditions. These materials are introduced to ease the understanding of the iterative techniques and implementation constraints encountered in the sections that follow.

4.2.1 The Poisson Equation

We wish to solve the Poisson equation on a d-dimensional domain Ω with boundary $\partial\Omega$. Denoting u as the solution of the Poisson equation, it satisfies

$$\boldsymbol{\nabla} \cdot (\boldsymbol{\nabla} u) = \nabla^2 u = -f \quad \text{on} \quad \Omega \subset \mathbb{R}^d, \tag{4.1}$$

where ∇^2 is the Laplacian operator and f is given. The Laplacian operator applied to u is defined as the divergence ($\boldsymbol{\nabla}\cdot$) of the gradient of u ($\boldsymbol{\nabla} u$). It can be expressed as the sum of the second partial derivatives of u along each dimension d:

$$\nabla^2 = \sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2}, \quad \text{with} \quad \mathbf{x} = (x_1, x_2, ..., x_d)^T \in \Omega. \tag{4.2}$$

The elliptic equation (Equation (4.1)) calls for boundary conditions which can be of Neumann type:

$$\frac{\partial u}{\partial n} = g(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega_N, \tag{4.3}$$

where $\partial u/\partial n := \boldsymbol{\nabla} u \cdot \mathbf{n}$ is the normal derivative at the boundary $\partial\Omega_N$ with $\mathbf{n}$ pointing outside of the domain, or of Dirichlet type:

$$u = u_D(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega_D, \tag{4.4}$$

where $\partial\Omega_N$ and $\partial\Omega_D$ are distinct portions of $\partial\Omega$ such that $\partial\Omega = \partial\Omega_N \cup \partial\Omega_D$. This problem is represented schematically in Figure 4.1. In this chapter, we focus on Neumann-type boundary conditions. Other types of boundary conditions are discussed in Section 4.7.

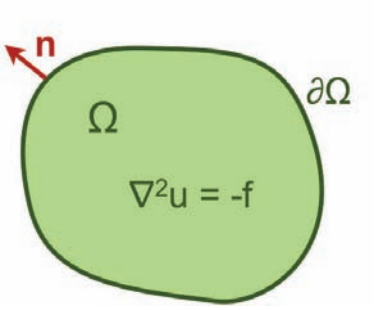


Figure 4.1. Poisson problem (Equation 4.1) on a 2D domain Ω with a boundary $\partial\Omega$. The vector $\mathbf{n}$ is normal to the boundary and points outside of the domain.

Note that for well-posed problems, when $\delta\Omega_D = \emptyset$, the data f needs to satisfy the compatibility condition

$$\int_\Omega f(\mathbf{x})d\mathbf{x} = \int_{\delta\Omega} g(\mathbf{x})d\mathbf{x}.$$

4.2.2 Finite Volume Discretization

In practice, Equation (4.1) has no close-form solution for general right-hand-sides f and domains Ω. In these cases, one has to rely on a numerical technique where an approximation of the solution u is sought in a finite-dimensional approximation space, by means of a discretization method, leading to a finite—although eventually large—set of equations. Classically, this discretization proceeds from a partition of the computational domain Ω into a finite number of control nodes, volumes, or elements. While several methods exist for the discretization of the Poisson equation, we select here the finite-volume (FV) method. Indeed, beside their intuitive nature and easy physical interpretation, the FV method is widely used in both computational fluid dynamics (CFD) and computer graphics communities, where correspondence between voxels and averages over FV cells is immediate. The popularity of the FV method also makes the availability of efficient solvers important, since the resolution of Poisson-like equations is a key ingredient of many CFD codes.

Roughly speaking, FV relies on the approximation of the average of the function u over the mesh cells (Figure 4.2). Specifically, the FV mesh is made of a set $\mathcal{T}$ of non-overlapping cells C_i covering Ω:

$$\Omega = \bigcup_{i\in\mathcal{T}} C_i, \quad C_i \cap C_j = \emptyset, \quad \forall i \neq j \in \mathcal{T}, \tag{4.5}$$

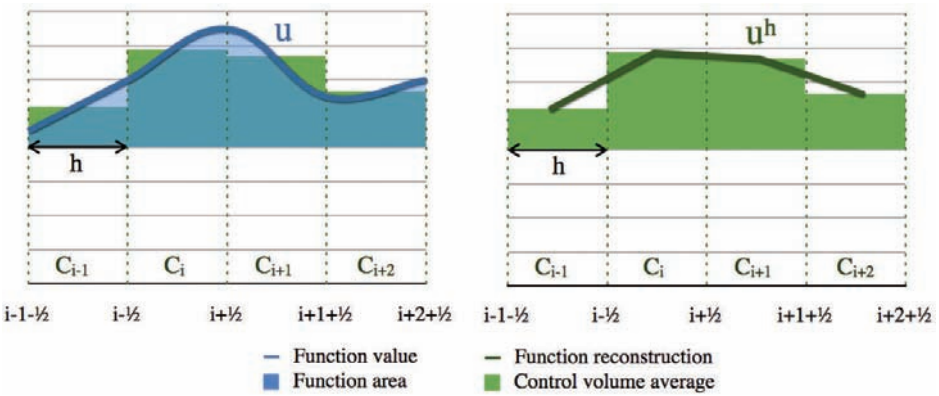


Figure 4.2. Cell values are computed by averaging the 1D function u with a FV method (left). The reconstruction of u^h by linear interpolation over the cell averages (right).

and we denote by u_i^h the computed average of u over the cell with index i:

$$u_i^h \approx \frac{1}{|C_i|} \int_{C_i} u \, dV. \tag{4.6}$$

The superscript h refers here to the discretized nature of the solution, h being related to the size of the cells.

Different types of meshes can be considered; we restrict ourselves to structured Cartesian grids made of cells with equal edge size h in all directions (the cell volume is h^d). For a sufficiently refined grid (i.e., small enough h), u_i^h can be identified with the value of u at the center $\mathbf{x}_i$ of cell C_i. In addition, such structured grids greatly simplify the reconstruction of the smooth approximate u^h from the averages arising from the multilinear interpolation between the cell centers. This reconstruction is schematically illustrated in Figure 4.2 (right).

Once the computational domain has been discretized into finite volumes, the objective is to derive a system of equations relating the values u_i^h for $i \in \mathcal{T}$. This is achieved by making use of Stokes' theorem [Spivak 71], which consists in replacing the integral of the Poisson equation over a cell by the integral of the normal flux $(\partial u/\partial n)$ over the cell's boundaries. Specifically,

$$\int_{C_i} \boldsymbol{\nabla} \cdot (\boldsymbol{\nabla} u) \, d\mathbf{x} = \int_{\partial C_i} \boldsymbol{\nabla} u \cdot \mathbf{n} \, d\mathbf{x} = - \int_{C_i} f \, d\mathbf{x} \simeq -f(\mathbf{x}_i) \, |C_i| \, . \tag{4.7}$$

Since $\boldsymbol{\nabla} u \cdot \mathbf{n}$ is the projection of the gradient of u in the normal direction $\mathbf{n}$ at a cell boundary, it reduces to the normal derivative $\partial u/\partial n$, where n takes value in $\{\pm x_i;\, i = 1, \ldots, d\}$, owing to the Cartesian structure of the mesh. Therefore, in

one dimension, for instance, Equation (4.7) reduces to

$$\left.\frac{\partial u}{\partial x}\right|_{x_i-h/2} - \left.\frac{\partial u}{\partial x}\right|_{x_i+h/2} = h f_i^h, \tag{4.8}$$

where $x_i \pm h/2$ are the locations of the cell interfaces and we have denoted $f_i^h = f(x_i)$. The FV system is finally obtained by substituting the fluxes $\boldsymbol{\nabla} u \cdot \mathbf{n}$ by their reconstructions from the set of averaged values $\left\{u_i^h; i \in \mathcal{T}\right\}$. Again, different reconstruction strategies can be used, and we adopt the second-order reconstruction, where the normal flux is based on the difference between the averages at the two cells C_i and C_j having in common an interface ∂C_{ij}:

$$\left.\frac{\partial u}{\partial n}\right|_{\partial C_{ij}} \approx \frac{u_i^h - u_j^h}{|\mathbf{x}_i - \mathbf{x}_j|}. \tag{4.9}$$

Inserting this approximation in the one-dimensional case of Equation (4.8), it becomes

$$\frac{u_i^h - u_{i-1}^h}{h} - \frac{u_{i+1}^h - u_i^h}{h} = \frac{2u_i^h - u_{i-1}^h - u_{i+1}^h}{h} \approx h f_i^h. \tag{4.10}$$

Similar expression can be immediately derived for higher-dimensional problems through tensorization, owing to the Cartesian nature of the grid. We only provide the case for $d = 3$, which corresponds to the discretization used in all subsequent development. In 3D, the cells are indexed by 3 subscripts (i, j, k) referring to the location of $C_{i,j,k}$ in the Cartesian grid ($C_{i,j,k}$ and $C_{i,j,k+1}$ are thus two neighboring cells in the third spatial direction). With this notation, the FV approximation of the Poisson equation over cell $C_{i,j,k}$ can be expressed as

$$\begin{aligned} 6u_{i,j,k}^h - u_{i-1,j,k}^h - u_{i+1,j,k}^h \quad & \cdots \\ \cdots - u_{i,j-1,k}^h - u_{i,j+1,k}^h \quad & \cdots \\ \cdots - u_{i,j,k-1}^h - u_{i,j,k+1}^h \quad &= \quad -h^2 f_{i,j,k}^h \end{aligned} \tag{4.11}$$

The discrete equation for cell $C_{i,j,k}$ involves the (unknown) averages over the cell and its six neighbors having a face in common (see Figure 4.3).

Writing this equation for all cells of the mesh, eventually using modified reconstructions of the flux for the cells neighboring $\partial\Omega$ (see discussion below), one ends with a system of $N = N_x \times N_y \times N_z$ equations for the cell averages $u_{i,j,k}^h$, $1 \le i \le N_x$, $1 \le j \le N_y$ and $1 \le k \le N_z$. This system can be rewritten in a matrix form as

$$A\mathbf{u} = \mathbf{f}, \tag{4.12}$$

where $A \in \mathbb{R}^{N \times N}$ is a sparse matrix, $\mathbf{u} \in \mathbb{R}^N$ is the vector containing the cell averages and $\mathbf{f}$ gathers the corresponding right-hand sides of Equation (4.11).

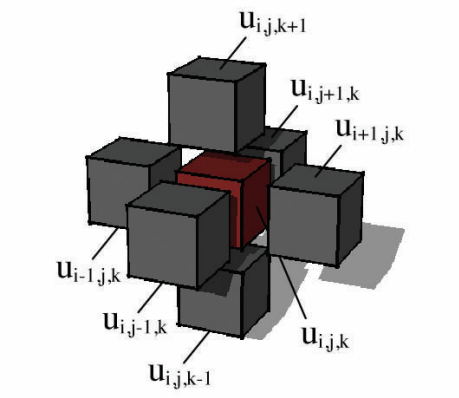


Figure 4.3. Enlarged view of the 3D Laplace stencil, the left-hand-side in Equation (4.11).

The sparsity of A arises from the fact that the fluxes are reconstructed from the immediate neighboring cells, such that each row of A has only seven nonzero entries as seen from Equation (4.11). As a result, the memory allocation for A is less than $7N$ (due to the treatment of the boundary conditions). An example of matrix A is shown in Figure 4.4 for $N_x = N_y = N_z = 3$, before reduction, due to boundary conditions.

$$
\begin{pmatrix}
6 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 6 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 6 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 6 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & -1 & 6 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & -1 & 6 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 6 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & -1 & 6 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 6 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 6 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 6 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 6 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 6 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 6 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 6 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 6 & -1 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 6 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 6 & -1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 6 & -1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 6 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 6 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 6
\end{pmatrix}
$$

Figure 4.4. Example matrix A for a $3 \times 3 \times 3$ discretization grid ($N = 27$) composed of 27^2 elements, where less than 7×27 of them are nonzero. This matrix results from the Laplace equation (Equation (4.11)), instantiated for each cell of $\mathcal{T}$.

4.2.3 Boundary Conditions (BC)

We choose for simplicity, the homogeneous Neumann condition (Equation (4.3)) on u along the domain boundary $\partial\Omega$,

$$\frac{\partial u}{\partial n} = 0, \quad \text{for } \mathbf{x} \in \partial\Omega. \tag{4.13}$$

This condition states that the flux (or normal derivative) of u is zero everywhere on $\partial\Omega$. In the context of potential flows, where u is the flow potential, this corresponds to no-through flow BC. For instance, in the classical projection-correction methods for solving the incompressible Navier-Stokes equations (see [Chorin 68]), such potential flow is used to enforce the divergence-free constraint on the velocity field.

In practice, ghost-cell techniques are commonly used to implement the homogeneous Neumann BC. It consists of creating a virtual layer of cells along the boundary, with values that mirror the inside domain. One of the interesting features of this ghost-cell approach is that it immediately extends to other types of BC (nonhomogeneous, Dirichlet, Fourier, periodic domains, etc.) making them very attractive in terms of general code implementation. Indeed, after defining the ghost-cells values (eventually updated at each iteration), the same stencil can be used for all the inner cells of the computational domain. In the case of the homogeneous Neumann BC, a ghost cell is taken equal to the inner domain cell sharing a face with it. Therefore, the flux between the two cells is zero (see Equation (4.9)). Other types of BC follow a similar procedure whereby the ghost-cell values are defined from their respective (inner domain) neighboring cell value (see discussion in [Patankar 80]).

4.3 Iterative Methods

Unless otherwise specified, the edge size of a cell is h. We also drop the cell index to alleviate the notation when representing the iteration number α as a subscript.

The sparse matrix A has important properties that can be exploited. First, A is *symmetric*, due to the symmetric definition of the fluxes between two neighboring cells (the flux going from a cell i to its neighbor j is the same as the flux from j to i); this symmetry also implies the conservative nature of the FV scheme. Second, A is *positive definite*. These properties make A invertible with a unique solution

$$\mathbf{u} = A^{-1}\mathbf{f}. \tag{4.14}$$

For small N, the matrix A can be inverted using a direct method, for example, Gaussian elimination or LU decomposition [Cormen et al. 01]. These classical methods are common but not very efficient because of their $O(N^3)$ complexity (recall that N is the number of unknowns to be solved in the system). In addition,

direct inversion methods consume a good deal of memory, because even if A is sparse, its inverse is usually full. As an example, the amount of memory required to compute and store A^{-1} for a medium-sized domain discretization, $N = 64^3$, in IEEE 754 single-precision floats would be superior to the capacity of current hardware: $(64^3)^2 \times \texttt{sizeof(float)} = 256$ GB.

Iterative methods have been developed to address this issue, because they can work with a matrix-free representation of the linear system. In these techniques, an approximation of the solution $\mathbf{u}$ is iteratively constructed through a sequence of vectors $\{\mathbf{v}_\alpha, \alpha = 0, 1, \ldots\}$ that converges to $A^{-1}\mathbf{f}$,

$$\lim_{\alpha \to \infty} \mathbf{v}_\alpha = \mathbf{u} = A^{-1}\mathbf{f}. \tag{4.15}$$

4.3.1 Simple Preconditioned Iterations

An immediate way to construct a convergent series of approximations is to rely on simple preconditioned iterations. Let P be an appropriate preconditioner of A (see examples below), such that $P^{-1}A$ has a lower condition number (stability to numerical operations) than A.

Let P be a preconditioner in the linear system $A\mathbf{u} = \mathbf{f}$, we can write

$$P\mathbf{u} = (P - A)\mathbf{u} + \mathbf{f} \quad \Leftrightarrow \quad \mathbf{u} = (I - P^{-1}A)\mathbf{u} + P^{-1}\mathbf{f}. \tag{4.16}$$

The smoothing iteration, derived from Equation (4.16), is the core of iterative methods and computes a new approximation $\mathbf{v}_{\alpha+1}$ from $\mathbf{v}_\alpha$ at iteration α,

$$\mathbf{v}_{\alpha+1} = (I - P^{-1}A)\mathbf{v}_\alpha + P^{-1}\mathbf{f}. \tag{4.17}$$

Let $\mathbf{e}_\alpha = \mathbf{u} - \mathbf{v}_\alpha$ be the error. We can subtract Equation (4.17) from (4.16) to obtain the error reduction at iteration α:

$$\mathbf{e}_{\alpha+1} = (I - P^{-1}A)\mathbf{e}_\alpha = M\mathbf{e}_\alpha \quad \text{thus} \quad \mathbf{e}^k = M^k\mathbf{e}_0, \tag{4.18}$$

where the iteration matrix M is multiplied with $\mathbf{e}_\alpha$ until a convergence condition $|\mathbf{e}_\alpha| < \epsilon$ is met (typical values for a $L2$ norm are taken below 10^{-3}).

A wide range of preconditioners is available. Equation (4.18) shows that a good preconditioner should be such that $P^{-1}A \simeq I$. On the other hand, Equation (4.17) shows that, to be applied, the iteration needs the calculation of the effect of P^{-1} on vectors: P should be easily inverted. These two concurrent features lead to the extreme preconditioners, $P = A$, which results in an exact error reduction in just one iteration, and $P = I$, which allows for trivial inversion, but may result in poor error reduction with the iterations (if converging at all).

To construct classical preconditioners, it is convenient to split the square matrix A into three parts:

$$A = L + D + U, \tag{4.19}$$

where L is the lower-triangular part of A, D its diagonal, and U the upper-triangular part of A, as illustrated in Figure 4.5.

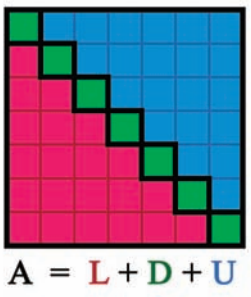


Figure 4.5. Decomposition of the matrix A into its lower-triangular part L, diagonal D, and upper-triangular part U, used in the construction of the preconditioners.

4.3.2 Jacobi Method

The first preconditioner P_J that we introduce is used in the pure iterative Jacobi method. It is defined as the diagonal part of A with $P_J = D$. Introducing P_J into Equation (4.17) gives the following Jacobi iteration:

$$\mathbf{v}_{\alpha+1} = (I - D^{-1}A)\mathbf{v}_\alpha + D^{-1}\mathbf{f}. \tag{4.20}$$

In the case of the 3D Laplace matrix (Figure 4.4), we observe that the diagonal is equal to 6 for the inner domain, therefore we can write $D = 6I$. Using this simplified form $D^{-1} = 1/6\, I$, the Jacobi iteration (4.20) becomes

$$\mathbf{v}_{\alpha+1} = (I - \frac{1}{6}A)\mathbf{v}_\alpha + \frac{1}{6}\mathbf{f}. \tag{4.21}$$

This method is also called the *method of simultaneous displacements*. As observed in the visual representation of the Jacobi iteration in Figure 4.6, all

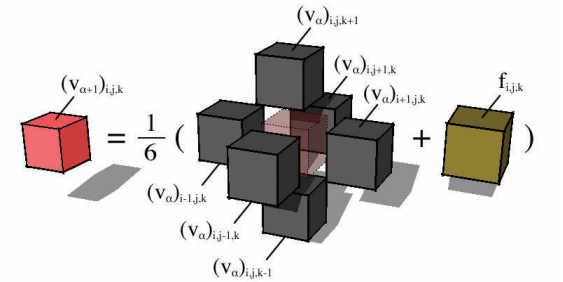


Figure 4.6. Visual representation of the Jacobi iteration. Note that the central cell of the stencil on $\mathbf{v}_\alpha$ is provided only as a visual cue and does not intervene in this equation because the diagonal of M is nullified ($I - 1/6D = 0I$).

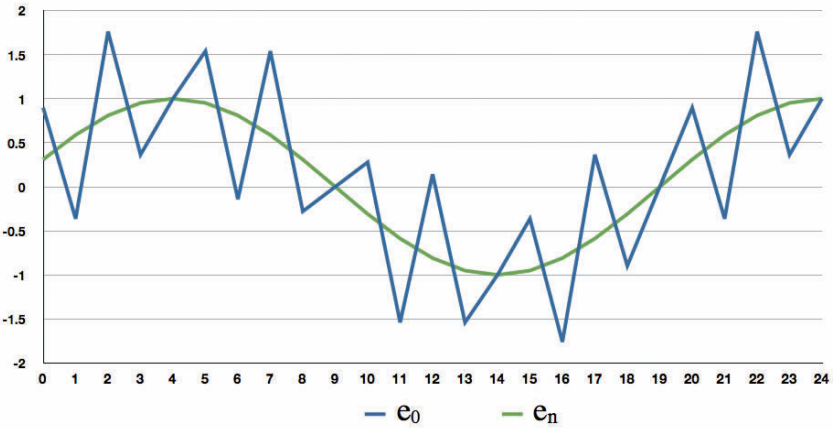


Figure 4.7. Result of several Jacobi iterations on a 1D error function. The high spatial frequency components, sharp edges of $\mathbf{e}_0$, are efficiently smoothed out after a few iterations while the low frequency components remain almost unchanged.

unknowns of $\mathbf{v}_{\alpha+1}$ can be simultaneously computed as each equation is independent and requires knowledge of only $\mathbf{v}_\alpha$ and $\mathbf{f}$.

The computational cost of one iteration is low, and the Jacobi method is straightforward to implement using two separate *ping-pong* buffers for $\mathbf{v}_\alpha$ and $\mathbf{v}_{\alpha+1}$. This method quickly suppresses the local components of $\mathbf{e}$ (with high spatial frequency) after a small number of iterations. Unfortunately, it does a poor job at suppressing the low-frequency components (global features of $\mathbf{e}$ spread across a distance wider than two consecutive cells) as illustrated on a 1D example in Figure 4.7.

The Jacobi method is widely used in the computer graphics community [Stam 99, Crane et al. 07] as it is easy to understand and implement. Despite its popularity, its cost of $O(N^2)$ iterations to reduce the error by a constant factor makes it impractical for problems requiring a good accuracy.

Section 4.5.2 covers the implementation of the Jacobi method using OpenCL.

4.3.3 Gauss-Seidel Method

Instead of considering only the diagonal of A, the Gauss-Seidel preconditioner P_{GS} is composed of its diagonal and lower-triangular part, $P_{GS} = L + D$. This makes P_{GS} closer to A than Jacobi's P_J, yet still easy to work with numerically. Introducing P_{GS} into Equation (4.17) gives the Gauss-Seidel iteration:

$$\mathbf{v}_{\alpha+1} = (I - (L + D)^{-1}A)\mathbf{v}_\alpha + (L + D)^{-1}\mathbf{f}. \tag{4.22}$$

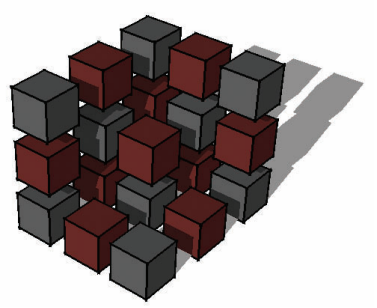


Figure 4.8. Red-black labeling of the grid cells for the Gauss-Seidel method. We observe that all neighbors of a red cell are painted in black and vice-versa, in order to prevent concurrent read and write access by two threads on the same cell.

The inclusion of the lower-triangular part L results in a dependency relation between unknowns in the linear system. The values for grid index (i, j, k) are thus being updated with new values coming from $\mathbf{v}_{\alpha+1}$ instead of $\mathbf{v}_{\alpha}$, for their neighbors numbered with lower indices $(i-1, j, k)$, $(i, j-1, k)$ and $(i, j, k-1)$. This intuitively leads to an improvement in the error reduction with the iteration. In practice, about half the number of Jacobi iterations is needed for a given reduction of the error.

This method is also called *method of successive displacements* because of the dependency between the updated neighbor cells. From an implementation point of view, the Gauss-Seidel method requires a single buffer to represent old and new values of $\mathbf{v}$ during an iteration, which leads to a concurrency problem when neighbor cells are read and written at the same time by different threads. A simple solution is to use a *red-black* ordering of the cells where each cell is affected by a color just like on a 3D checkerboard [Strang 07], as illustrated in Figure 4.8.

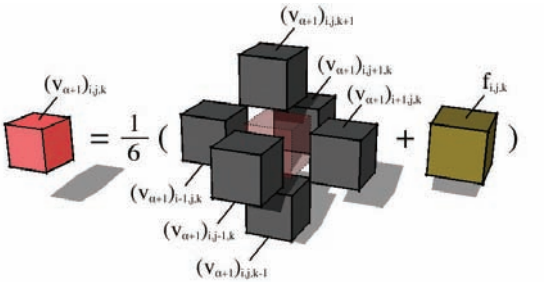


Figure 4.9. Visual representation of the second half of the Gauss-Seidel iteration. Please note that the black neighbors of the red cell have already been computed during the first half iteration on black cells.

The Gauss-Seidel iteration is then split into two steps where the black cells are updated first because they do not have any face in common, and the red cells are updated using the newly computed values of the black cells, as illustrated in Figure 4.9.

Section 4.5.3 covers the implementation of the Gauss-Seidel method using *red-black* ordering with OpenCL.

4.3.4 Successive Over-Relaxation (SOR)

Observing the limitations of the two previous preconditioners and their $O(N^2)$ computational complexity, one wonders if it is possible to over-correct the error and reduce the low-frequency features by extrapolating the local correction provided by each iteration.

We introduce the weighting factor ω to control the amount of overshooting applied to the previous preconditioners. When combined with the Gauss-Seidel preconditioner, this method is called SOR and can be written as

$$\mathbf{v}_{\alpha+1} = (D + \omega L)^{-1}(\omega \mathbf{f} - (\omega U + (\omega - 1)D)\mathbf{v}_\alpha). \tag{4.23}$$

This preconditioner is proven to converge when ω takes value between 0 and 2, but we are more interested in a fast convergence rather than just convergence. Using a weight factor of 1 is equivalent to the standard Gauss-Seidel method, while factors greater than 1 lead to an over-relaxation of the smoothing correction, which results in a faster propagation of low-frequency error components on the grid. The optimal factor ω_{opt}, for which the convergence speed is the fastest, depends on the spectral radius ρ of the iteration matrix M, or the maximum absolute of the eigenvalues λ of this matrix,

$$\rho(M) = \max|\lambda(M)|, \qquad \omega_{opt} = \frac{2}{1 + \sqrt{(1 - \rho^2(M))}}.$$

Finding the maximal eigenvalue of M has the same complexity as computing A^{-1}, which is why good values of ω are often empirically determined. The value $2/3$ is a safe estimation for ω on small domains, while optimal values tend to asymptotically get closer to 2 when the solution domain grows (see Chapter 6.2 of [Strang 07]). Taking a value higher than ω_{opt} usually results in bad performance, because the approximation convergence tends to oscillate.

Section 4.5.4 covers the implementation of the SOR method based on forward Gauss-Seidel with OpenCL.

Other preconditioners do exist and become more efficient as they grow in numerical and understanding complexity. In the next section, instead of an exhaustive review of preconditioners, we address the inherent problem of iterative methods, which is the reduction of the low spatial frequency components. A comprehensive review with a deeper mathematical analysis of preconditioners is available in Saad's and Strang's reference books [Saad 03, Strang 07].

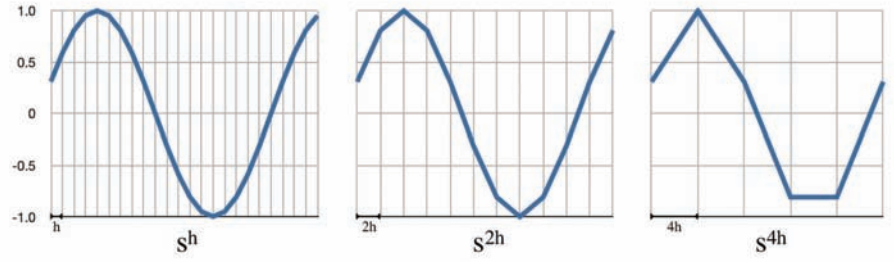


Figure 4.10. Sampling of a smooth function at different resolution levels h (left), $2h$ (center), and $4h$ (right). We observe that the low-frequency component represented on the finest level (h) becomes rougher as the grid gets coarser ($2h$, $4h$).

4.4 Multigrid Methods (MG)

We have observed the good performance of pure iterative methods to smooth out high-frequency components, where error features are spread over a $1h$ distance. Unfortunately, after few iterations, the error reduction per iteration decreases because the remaining error now contains only low frequencies that cannot be efficiently smoothed out.

Southwell [Southwell 35] introduced a method where a second grid helps to maintain a fast convergence: the problem is discretized a second time on a grid with cells of $2h$ edge size in order to smooth out the high frequencies (now spread over a $2h$ distance) using existing preconditioners, as illustrated in Figure 4.10. This concept was later generalized to a new class of multiscale iterative methods where the initial problem is solved on grids of different resolutions.

4.4.1 Multigrid Correction Scheme

The MG correction scheme (CS) method is the most appropriate MG method available to solve partial differential equations with linear coefficients as in the Poisson equation discretized with FV on a uniform grid. Like most MG methods, the smoothing iterations reduce the high-frequency error. The CS accelerates the reduction of the lower frequencies by rescaling the error on a coarser grid.

A typical CS iteration is shaped into a recursive V-cycle, illustrated in Figure 4.11, which can be divided into substeps relying on five different operators:

- S smoothing iteration,
- R residual computation,
- P fine to coarse projection,
- I coarse to fine interpolation,
- C approximation correction.

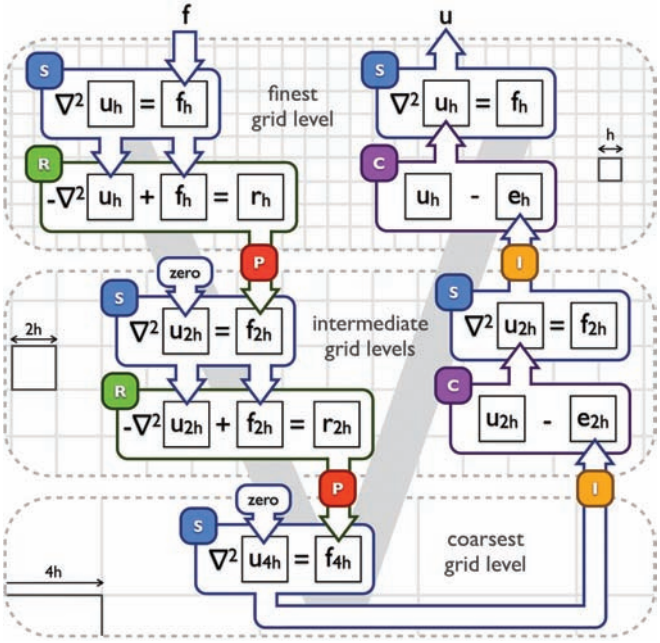


Figure 4.11. Three-level CS MG V-cycle illustrating the connection between the smoothing, residual, projection, interpolation, and correction operators.

Let us illustrate this process by walking through the different steps of a CS iteration.

Any smoothing iteration S from pure iterative methods can be used and is consecutively applied μ_{pre} times to begin the V-cycle. Its role is not to solve the problem, but to smooth the error until its high-frequency components are almost suppressed. This leads to an error $\mathbf{e}_\alpha^h$ composed only of features spread across more than two consecutive cells of size h. Computing the correction term to remove all the remaining components for the error would require the knowledge of the unknown solution $\mathbf{e}_\alpha^h = \mathbf{u}^h - \mathbf{v}_\alpha^h$. Instead, we can compute the residual of $\mathbf{v}_\alpha^h$ using R:

$$\mathbf{r}_\alpha^h = \mathbf{f}^h - A^h \mathbf{v}_\alpha^h \tag{4.24}$$

Reorganizing Equation (4.24) leads to the residual Poisson problem,

$$A^h \mathbf{e}_\alpha^h = \mathbf{r}_\alpha^h. \tag{4.25}$$

Knowing that we almost suppressed all the high frequencies from $\mathbf{r}_\alpha^h$, we observe that the remaining smooth function can be represented on a coarser grid

(where the edge size becomes $2h$) without losing important information. The projection operator P transfers $\mathbf{r}_\alpha^h$ from a fine -h-spaced- grid to a coarser—$2h$-spaced—grid producing a rougher function $\mathbf{f}_\alpha^{2h}$. This projection actually leads to a new Poisson problem defined on a coarser grid, where all remaining frequencies are a bit higher. The approximated solution to this problem $\mathbf{v}_\alpha^{2h}$ is *not* a projection of $\mathbf{v}_\alpha^h$, but the correction term required to reduce higher error frequencies on $\mathbf{v}_\alpha^h$:

$$A^{2h}\mathbf{v}_\alpha^{2h} = \mathbf{f}_\alpha^{2h}. \tag{4.26}$$

The smoothing-residual-projection step is then repeated to transfer the problem to coarser grids and reduce lowest frequency components. The descent is stopped when the resulting domain grid is coarse enough for the problem to be directly solved, using, for example, a standard elimination method. As the grid is particularly coarse and small at this point, we later choose to apply S several times to reduce the remaining (fairly high) frequencies.

Once Equation (4.25) has been solved on the coarsest grid, the resulting solution $\mathbf{e}_\alpha^{2^z h}$ is the complement to the correction term required to suppress the lowest spatial frequency error components. It can then be added to the approximation of the finer resolution $\mathbf{v}_\alpha^{2^{z-1}h}$, in order to correct both the lowest and second-lowest frequency errors. This ascending step of the V-cycle is then recursively applied until the approximation $\mathbf{v}_\alpha^h$ is reached on the finest grid. The interpolation operator I is used to transport this correction term from a coarse grid to a finer one. This operator is coupled with the correction operator C, a simple vector subtraction on the finer level:

$$\mathbf{v}_{\alpha+1}^{2^{z-1}h} = \mathbf{v}_\alpha^{2^{z-1}h} - I_{2^z h}^{2^{z-1}h}\mathbf{v}_\alpha^{2^z h}, \tag{4.27}$$

where $I_{2^z h}^{2^{z-1}h}$ is the matrix of I, interpolating a vector of $2^z h$-width cells to a vector of $2^{z-1}h$-width cells.

By solving the residual Poisson equation on the coarsest grid, we obtain the error correction term that we need to apply to the $\mathbf{v}_\alpha^h$ approximation in order to minimize the lowest frequency of the residual on the finer grid. In order to reduce the residual on the finer grids, we recursively interpolate and correct these terms by chaining I and C on finer grids. This process corrects the current approximation for the finer grid, but also adds higher-frequency components due to small information loss during the projection. In order to reduce these, we apply S μ_{post} times.

While many projection and interpolation schemes exist to transport discretized functions between grids of different resolution, we choose trilinear interpolation as it remains consistent with FV methods (see Figure 4.12). As we later observe during the implementation, it is also one of the most computationally efficient schemes using OpenCL's hardware-accelerated multilinear filtering.

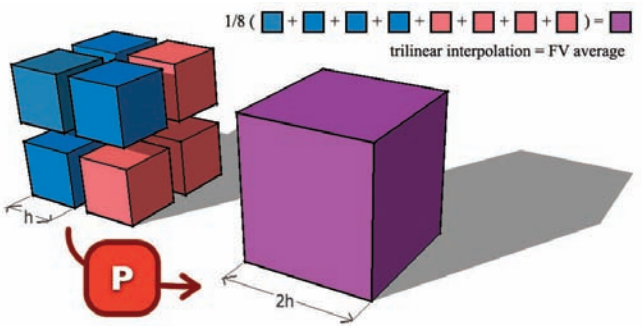


Figure 4.12. Trilinear interpolation process approximating the FV average, used in the OpenCL projection kernel.

Smoothing iterations at coarser grid levels (where N is recursively divided by eight for each projection) are computationally cheaper than iterations on the finest grid. In our implementation, we decide not to use a simple V-cycle but a multi-W-cycle. In that case, the first interpolation ascent is stopped before reaching the finest level and descends again toward the coarsest level to further reduce low-frequency errors with minimal computational cost. After a few of these intermediate cycles, the correction is finally interpolated to the finest level. In fact, we increase up to four subcycles to take advantage of this property, to achieve a faster convergence rate without increasing the computational cost too significantly. Finally, CS MG iterations are repeated until a convergence criterion ($|\mathbf{e}_\alpha| < \epsilon$) is met.

An extended overview of other MG methods and their applications can be found in McCormick's and Briggs' reference books [McCormick 88,Briggs et al. 00].

4.5 OpenCL Implementation

4.5.1 Overview

OpenCL is an open standard framework for programming heterogeneous parallel computers like multicore CPUs and GPUs. Like OpenGL, its specification is managed by the Khronos Group, which is composed of hardware and software industry leaders. The last revision of the OpenCL 1.0 specification [Khronos Group 09] is available online on the OpenCL registry website (http://www.khronos.org/registry/cl), along with C++ host bindings (`cl.hpp`) which greatly simplify the host API setup and communication calls.

The OpenCL framework is split into two parts. First, a host API, running on the CPU to initializes the devices and memory buffers, controls the execution of compute kernels and enqueues buffer exchanges between the host and devices, in

the same way as the traditional OpenGL/GLSL shader and texture setup. Second, *compute kernels* written in the OpenCL C language, are programs executed in parallel on the devices in the same way as traditional vertex or pixel shaders but with greater flexibility to address broader problems.

The OpenCL API specifies two kinds of memory objects: *buffers* and *images*. Buffers are contiguous arrays of memory indexed by 1D coordinates and composed of any available type (`int, float, half, double, int2, int3, int4, float2, ...`). These can either be allocated in *global*, *local* or *constant* memory. Global memory can be shared between the host and the devices by enqueuing read or write commands to exchange these data; it is abundant but it has a high latency and must be used with caution, while local memory is faster but has a very limited size. Coalesced memory accesses can reduce the latency, but when memory read and write patterns are random, images can be used to mitigate this latency. Images share many similarities with textures: they support an automatic caching mechanism, their access can be filtered through a *sampler* with multilinear interpolation, and out of bounds access behavior can be configured.

We make use of *images* whenever possible because of two reasons. First, our memory access patterns are mostly random. Second, the sampler filtering can greatly reduce the computational costs of certain operations such as multilinear interpolation in the projection operator, or automatic clamping of image coordinates to handle ghost cells with a homogeneous boundary condition.

Finally, parallelization is achieved by enqueueing the execution of compute kernels over *work-groups* or ranges of threads organized in 1, 2, or 3D. Each work-group is composed of *work-items*, or threads indexed by a unique 1, 2, or 3D identifier inside the global *work-group* range.

Initializing the compute devices is straightforward using the C++ host bindings. We first initialize the platform to access the underlying compute devices and select either CPU devices, GPU devices, or both types. Then we create a context and a command queue in order to execute compute kernels and enqueue memory transfers.

```
// fetch all GPU devices on the first OpenCL platform
std::vector<cl::Platform> platforms;
std::vector<cl::Device> devices;
cl::Platform::get(&platforms);
platforms.at(0).getDevices(CL_DEVICE_TYPE_GPU, &devices);

// setup a context and a command queue for the first device
cl::Context context(devices);
cl::CommandQueue queue(context, devices.at(0));
```

Finally, compute kernels are loaded and compiled from OpenCL source:

```
// load an OpenCL source file into a std::string
std::ifstream srcfile("kernels.cl");
```

```
std::string src(std::istreambuf_iterator<char>(srcfile),
                std::istreambuf_iterator<char>(0));

// compile the device program and load the "zero" kernel
cl::Program program(context, cl::Program::Sources(1,
              std::make_pair(src.c_str(), src.size())));
program.build(devices, "-Werror");
cl::Kernel kzero = cl::Kernel(program, "zero");
```

This *zero* compute kernel is later used to clear global memory buffers on the device. With OpenCL's C language, it is written as follow:

```
kernel void zero(global float *vh)
{
  // id contains the 3D index of the current cell
  const float4 id = (float4)(get_global_id(0),
       get_global_id(1), get_global_id(2), 0);

  // sz contains the buffer stride along each 3D axis
  const float4 sz = (float4)(1, get_global_size(0),
    get_global_size(0) * get_global_size(1), 0); // strides

  // vh is a global buffer used to write the zero output
  vh[(int)dot(id, sz)] = 0; // vh[id.x+id.y*sz.y+id.z*sz.z]
}
```

This kernel must be launched with a global *work-size* equal to the number of elements in the input buffer `vh` so that each element is written by exactly one *work-item* (or thread). The `id` and `sz` variables are initialized with the 3D work-item identifier and work-size stride so that their dot product directly gives the corresponding memory location in the 1D buffer (i.e., `id.x + id.y*sz.y + id.z*sz.z`).

In order to run this compute kernel, we need to allocate a memory buffer on the device and create the global (computational domain) and local (concurrent work-items) work-size ranges:

```
// initialize a read-only buffer for 64^3 4-bytes floats,
// this buffer can not be update by the host, only read from
cl::Buffer buffer(context, CL_MEM_READ_ONLY, 64*64*64 * 4);

// prepare the work-size ranges and kernel arguments
cl::NDRange gndr(64, 64, 64), lndr(8, 8, 8);
zero.setArg(0, buffer); // bind the buffer to vh

// enqueue the kernel and wait for it to finish its task
queue.enqueueNDRangeKernel(zero, cl::NullRange, gndr, lndr);
queue.enqueueBarrier();
```

Finally, data can be retrieved on the host by enqueueing a read command to the command queue,

```
float data[64 * 64 * 64];

// blocking read of the buffer from 0 to 64*64*64 into data
queue.enqueueReadBuffer(buffer, CL_TRUE, 0, 64*64*64, data);
```

Three-dimensional images are allocated in almost the same way as buffers. They require additional knowledge of the data type (`int, float, ...`) for interpolation and x-, y-, and z-extents for spatial caching,

```
// one-component float image format
cl::ImageFormat fmt(CL_R, CL_FLOAT);
cl::Image3D img(context, CL_MEM_READ_ONLY, fmt, 64, 64, 64);
```

Before implementing iterative solvers with OpenCL, we define a structure to contain the required elements of a Poisson problem (size, input function f, approximation v, and residual r):

```
typedef struct {
  cl::size_t<3> size; // 3D size of the problem
  cl::Image3D fh  // discretized f function
  cl::Image3D vh; // solution approximation
  cl::Image3D rh; // residual
} Problem;
```

4.5.2 Jacobi Method

In the Jacobi method, each line of the linear system is computed independently from the others. This results in an easy parallelization of the code. In theory, the cost of this advantage is the need to allocate two buffers to store $\mathbf{v}$: one is used as an input for the smoothing iteration Equation (4.20), $\mathbf{v}_\alpha$, and the other to write the result $\mathbf{v}_{\alpha+1}$. In practice, we use a scratch buffer for all kernel outputs leading to no overhead, compared with other methods.

Because we have to access the neighbors of each cell (Figure 4.3), coalescent memory access is not achievable using the device's global memory. In order to overcome this limitation, we use OpenCL *images* which are in many ways similar to textures in classical GPU programming. Furthermore, 3D images have an automatic caching mechanism which greatly accelerates the memory access pattern encountered in smoothing iterations.

In order to satisfy homogeneous conditions on the boundary of the solution domain, access to the value of neighbors located on ghost cells outside of the domain is clamped to the edge of the boundary (and therefore of the *image*) in order to automatically copy the corresponding value on the inside.

First, we define offsets in each dimension to access the cells' neighbors,

```
#define dx (float4)(1, 0, 0, 0)
#define dy (float4)(0, 1, 0, 0)
#define dz (float4)(0, 0, 1, 0)
```

The fourth dimension of the `float4` struct is never used here but is required to specify image sampling coordinates. The Jacobi compute kernel for the device is then implemented as follows:

```
// fh and vh are input images containing the f and v values,
// vvh is an output buffer where the new value of v is
// written and h2 is the cell width h squared
kernel void jacobi(read_only image3d_t fh,
                   read_only image3d_t vh,
                   global float *vvh, float h2)
{
  const float4 id = (float4)(get_global_id(0),
          get_global_id(1), get_global_id(2), 0);
  const float4 sz = (float4)(1, get_global_size(0),
       get_global_size(0) * get_global_size(1), 0);

  // sampler for accessing the vh and fh images,
  // out of bounds accesses are clamped to the domain edges
  const sampler_t sampler = CLK_ADDRESS_CLAMP_TO_EDGE;

  const float s =
    (read_imagef(vh, sampler, id-dx).x +
     read_imagef(vh, sampler, id+dx).x +
     read_imagef(vh, sampler, id-dy).x +
     read_imagef(vh, sampler, id+dy).x +
     read_imagef(vh, sampler, id-dz).x +
     read_imagef(vh, sampler, id+dz).x -
     h2 * read_imagef(fh, sampler, id).x) / 6.0f;

  vvh[(int)dot(id, sz)] = s;
}
```

The function `read_imagef` is a built-in function which accesses a `read_only` image through a sampler at a specific coordinate, passed as a `float4` vector, and returns a `float4` vector containing the result. Since we initialize `fh` and `vh` as one component image, only the first component (`x`) of the result is meaningful.

This kernel is launched with a global work size equal to the 3D extents of the domain grid. The local work size depends on the capabilities of the OpenCL compute device and must be a divider of the global work-group size along each dimension. Experience shows that a cubic size (and in particular (8, 8, 8) for current GPUs) is an optimal work-group configuration because it leads to a minimal spatial scattering of memory accesses, thus fully exploiting the images cache. After each iteration, the output buffer is copied back to the `vh` image to be reused,

using the host API:

```
// offset and sz are size_t[3], offset contains zeros
// and sz contains the Problem size or 3D image extents
queue.enqueueCopyBufferToImage(buffer, image, 0, offset, sz);
queue.enqueueBarrier();
```

Once every few iterations, the approximation error of $\mathbf{v}_\alpha$ is tested on the host to decide wether to continue refining or not by computing the L^2 norm of the residual on the host and comparing it against an ϵ value:

```
// compute the residual for the current Problem p
residual(p.fh, p.vh, p.rh, p.size, h2);
queue.enqueueReadImage(p.rh, CL_TRUE, nullsz,
                       p.size, 0, 0, &r[0]);
float rnorm = L2Norm(r, fine.size); // sqrt(sum(r*r))

// break the solver loop
if(rnorm < epsilon) break;
```

4.5.3 Red-Black Gauss-Seidel Method

This kernel is very similar to the Jacobi kernel, the only remarkable difference in this implementation being the red-black ordering, which accelerates the theoretical convergence rate by a factor of two, where `red` is set to either one or zero, respectively with the current red-black pass type.

```
kernel void rbgs(read_only image3d_t fh,
                 read_only image3d_t vh,
                 global float *vvh,
                 float h2, int red)
{
  // the x cell identifier is multiplied by two
  // only work on either red or black cells
  float4 id = (float4)(get_global_id(0) << 1,
       get_global_id(1), get_global_id(2), 0);
  const float4 sz = (float4)(1, get_global_size(0),
         get_global_size(0)*get_global_size(1), 0);
  const sampler_t sampler = CLK_ADDRESS_CLAMP_TO_EDGE;

  // the initial x cell identifier offset depends on the
  // parity of id.y+id.z and on the current pass color
  id.x += ((int)(id.y + id.z + red) & 1);

  ... // compute s (see Jacobi)

  vvh[(int)dot(id, sz)] = s;
}
```

This kernel is launched twice to perform a full Gauss-Seidel iteration, each time with a global work size equal to the solution domain grid extents but halved on the first dimension to account for the red-black interleaving computed in `id` so that only half of the cells are accessed. As in the Jacobi implementation, the output buffer is copied to the image after each kernel call or half-iteration.

4.5.4 Successive Over-Relaxation

Implementation of SOR is a trivial addition to the Gauss-Seidel kernel, the only difference being the specification of ω, used as a weight factor for error over-correction to accelerate the convergence rate by reducing lower frequency error components faster than the two previous methods.

```
kernel void rbsor(..., float w) // weighting factor w
{
  ... // compute id and sz (see Red-Black Gauss-Seidel)

  vvh[(int)dot(id, sz)] =
    (1 - w) * read_imagef(vh, sampler, id).x + w * s;
}
```

4.5.5 Multigrid Correction Scheme

The host part of the MG CS method exactly mirrors the V-cycle presented in Figure 4.11. It is split into three steps. First, we have a descending step: high frequencies are reduced with iterations of S, then the residual is computed using R and projected to the next coarser grid using P until the coarsest grid is reached. Second, the coarsest grid is solved by applying multiple iterations during S until the coarsest problem (8^3) is almost solved. Finally, during the ascending step, the correction is interpolated back (C + I) and smoothed on the finer grids using S until the finest level is reached again. Actual calls to the respective compute kernels are encapsulated into helper functions, which take care of buffer and range initializations for simplicity purposes.

```
std::vector<Problem> p; // allocate a Problem for each level
init_problems(); // and reduce its size accordingly until
                 // the coarsest level is reached

// V-cycle descending step, from finest to coarsest level
for(k = 0; k < int(p.size()-1); ++k)
{
  rbsor(p[k].fh, p[k].vh, p[k].size, h2, 0.75f, preSteps);
  residual(p[k].fh, p[k].vh, p[k].rh, p[k].size, h2);
  project(p[k].rh, p[k+1].fh, p[k+1].size);
  zero(p[k+1].vh, p[k+1].size);
}
```

```
// "Direct" solving on the coarsest level
rbsor(p[k].fh, p[k].vh, p[k].size, h2, 1.5f, directSteps);

// V-cycle ascending step, from coarsest to finest level
for(--k; k >= 0; --k)
{
  interpolate_correct(p[k+1].vh, p[k].vh, p[k].size);
  rbsor(p[k].fh, p[k].vh, p[k].size, h2, 1.25f, postSteps);
}
```

The W-cycle is a direct extension of this code, adding an inner loop for $k_{\max} > k > 0$ in order to repeat the subcycle several times before finally reaching the finest level. Experimentations show that choosing two pre-smoothing passes, four post-smoothing at each level, and 32 direct-smoothing iterations for the coarsest level leads to the best measured convergence rate. Additionally, using four subcycles greatly reduces the overall computation time and seems to be the best configuration for medium to large grid sizes ($\geq 64^3$).

4.5.6 Residual, Projection and Interpolation Operators

In the same fashion as the smoothing operator, the residual operator R is a direct translation of the residual equation (Equation (4.24)) into parallel code. Input images are $\mathbf{f}^h$ and $\mathbf{u}^h$, and $\mathbf{r}^h$ is an output buffer to be later copied into an image in order to be projected using multilinear interpolation.

```
void kernel residual(read_only image3d_t fh,
                     read_only image3d_t vh,
                     global float *rh)
{
  ... // compute id and sz (see Jacobi)
  const sampler_t sampler = CLK_ADDRESS_CLAMP_TO_EDGE;

  rh[(int)dot(id, sz)] =
       - read_imagef(fh, sampler, id).x -
    (6 * read_imagef(vh, sampler, id).x -
        (read_imagef(vh, sampler, id-dx).x +
         read_imagef(vh, sampler, id+dx).x +
         read_imagef(vh, sampler, id-dy).x +
         read_imagef(vh, sampler, id+dy).x +
         read_imagef(vh, sampler, id-dz).x +
         read_imagef(vh, sampler, id+dz).x)) / h2;
}
```

To implement the projection operator, P, we take advantage of the image-filtering capabilities offered by OpenCL, resulting in a tremendous computation acceleration when dedicated hardware is present, like texture units on GPUs. We

use trilinear interpolation (Figure 4.12) to average eight cells on the fine level into one cell on the coarser level by enabling linear filtering and taking a sample at the center of the eight fine cells:

```
kernel void project(read_only image3d_t rh,
                    global float *f2h)
{
  ... // compute id and sz (see Jacobi)

  // filter images with trilinear interpolation:
  // cell centers indexing begins at 0.5 so that
  // integer values are automatically interpolated
  const sampler_t sampler = CLK_FILTER_LINEAR;

  // make the image coordinate at the vertex shared
  // between the eight finer grid (see fig. 1.11)
  // then multiply by 4 for coarsening: (2h)^2 = 4 h^2
  f2h[(int)dot(id, sz)] =
    read_imagef(rh, sampler, id * 2 + dx+dy+dz).x * 4;

}
```

In order to avoid redundant copying of buffers to images, we decide to combine the interpolation I and correction C operators into one kernel, where the interpolated correction is directly added to the finer grid level in one pass:

```
// interpolate v2h and correct vh to reduce low freqs.
kernel void interpolate_correct(read_only image3d_t v2h,
                                read_only image3d_t vh,
                                global float *vh)
{
  ... // compute id and sz (see Jacobi)
  const sampler_t sampler = 0;

  vh[(int)dot(id, sz)] = read_imagef(vh, sampler, id).x -
                         read_imagef(v2h, sampler, id/2).x;
}
```

4.6 Benchmarks

As expected, we can observe in Figure 4.13 an exponential number of iterations required for the Jacobi as its complexity reaches $O(N^3)$. The Gauss-Seidel method has the same complexity but requires fewer iterations as the constant complexity factor is halved.

The SOR method dramatically reduces this factor by over correcting the local error, but its complexity is still exponential. Fortunately, the CSMG method is confirmed to have a linear complexity of $O(N)$, where N is the number of unknowns or cells.

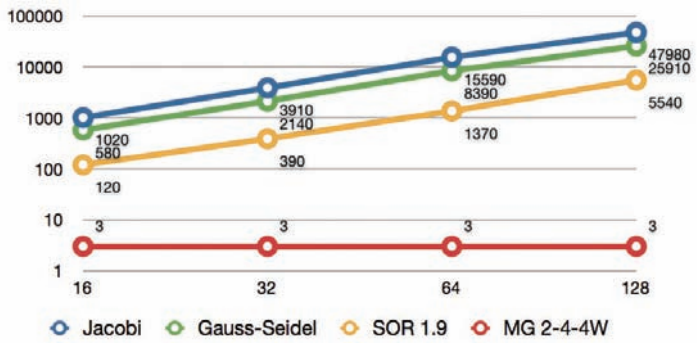


Figure 4.13. Iterations per method for cubic domains until $|\mathbf{e}_\alpha| < 10^{-3}$. The X-axis represents the domain size (cubed) and the Y-axis shows the number of iterations required to converge to $\epsilon = 10^{-3}$.

Although its iterations have a higher computational cost, the multigrid correction scheme method shows a clear advantage over the pure iterative methods in terms of computation time per unknown in Figure 4.14. The setup cost of the CSMG method makes it more efficient for large problems than smaller ones ($< 32^3$) where the SOR method should be preferred.

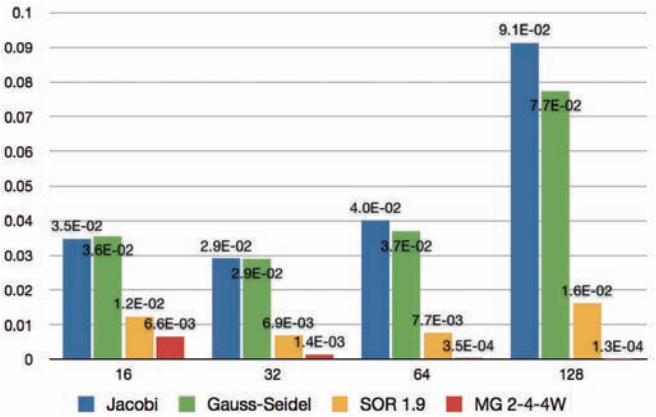


Figure 4.14. Computation time (μs) per cell for cubic domains on a GPU. The X-axis represents the domain size (cubed) and the y-axis shows the computation time per cell to converge to $\epsilon = 10^{-3}$.

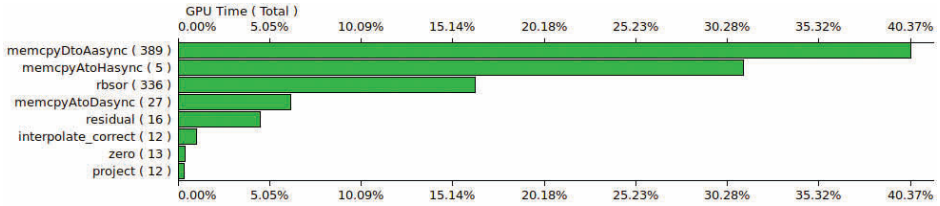


Figure 4.15. Time profiling of the execution of four CSMG 4W-cycles for a 128^3 computational domain, running on a Nvidia GTX-285 GPU. More than half of the time is spent either copying buffers into images (`memcpyDtoAasync`) or transfering the residual to the host (`memcpyAtoHasync`) to test the convergence.

4.7 Discussion

We introduced the theoretical background and implementation framework for a fast OpenCL solver for the 3D Poisson equation with Neumann external boundary condition. This is by no means a generic solver, but it can be extended to address other problems such as different boundary conditions or the discretization method.

In particular, writing to OpenCL images results in a significant computation-time decrease; for the current implementation, half of the time is spent copying output buffers back into images (see Figure 4.15). Unfortunately, this extension would alienate most of the current OpenCL hardware because writing to 3D images is an extension supported by very few devices as of the writing of this book.

Finally, using a parallel reduction on the OpenCL device to compute the residual norm would also result in a significant performance boost. Indeed, it would require transfering only one float value instead of the whole residual grid to test convergence on the host and decide whether or not to continue refining the solution approximation.

Bibliography

[Briggs et al. 00] William L. Briggs, Van Emden Henson, and Stephen F. McCormick. *A Multigrid Tutorial*, Second edition. Philadelphia: SIAM Books, 2000.

[Chorin 68] Alexandre J. Chorin. "Numerical Solution of the Navier-Stokes Equations." *Mathematics of Computation* 22:104 (1968), 745–762.

[Cormen et al. 01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. New York: McGraw-Hill Higher Education, 2001.

[Crane et al. 07] Keenan Crane, Ignacio Llamas, and Sarah Tariq. "Real-Time Simulation and Rendering of 3D Fluids." In *GPU Gems 3*, edited by Hubert Nguyen, Chapter 30. Reading, MA: Addison Wesley Professional, 2007.

[Khronos Group 09] Khronos Group. *The OpenCL Specification,* version 1.0.48. Khronos OpenCL Working Group, 2009. Available online (http://khronos.org/registry/cl/specs/opencl-1.0.48.pdf).

[McCormick 88] Stephen F. McCormick. *Multigrid Methods: Theory, Applications, and Supercomputing.* New York: Marcel Dekker, 1988.

[Patankar 80] Suhas V. Patankar. *Numerical Heat Transfer and Fluid Flow.* New York: Hemisphere Publishing Corporation, 1980.

[Saad 03] Youssef Saad. *Iterative Methods for Sparse Linear Systems*, Second edition. Philadelphia: Society for Industrial and Applied Mathematics, 2003.

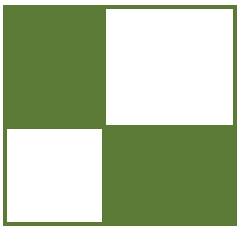
[Southwell 35] Richard V. Southwell. "Stress-Calculation in Frameworks by the Method of Systematic Relaxation of Constraints. I and II." In *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, pp. 56–96. London, 1935.

[Spivak 71] Michael Spivak. *Calculus on Manifolds: A Modern Approach to Classical Theorems of Advanced Calculus.* New York: HarperCollins Publishers, 1971.

[Stam 99] Jos Stam. "Stable Fluids." In *SIGGRAPH '99: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 121–128. New York: ACM Press/Addison-Wesley Publishing Co., 1999.

[Strang 07] Gilbert Strang. *Computational Science and Engineering.* Wellesley, MA: Wellesley-Cambridge Press, 2007.

[Yu et al. 04] Yizhou Yu, Kun Zhou, Dong Xu, Xiaohan Shi, Hujun Bao, Baining Guo, and Heung-Yeung Shum. "Mesh Editing with Poisson-Based Gradient Field Manipulation." *ACM Transactions on Graphics* 23:3 (2004), 644–651.



Contributors

Kristof Beets is the business development manager for POWERVR Graphics at Imagination Technologies. In this role he leads the overall graphics business promotion and technical marketing efforts, the in-house demo development team. Previously he managed the POWERVR Insider ecosystem and started work as a development engineer on SDKs and tools for both PC and mobile products as a member of the POWERVR Developer Relations Team. Kristof has a first degree in electrical engineering and a master's degree in artificial intelligence, both from the University of Leuven, Belgium. He has spoken at numerous industry events including SIGGRAPH, GDC, and Eurographics, and has had articles published in *ShaderX2*, *5*, *6*, and *7* and *GPU Pro* books as well as online by the Khronos Group, Beyond3D, and 3Dfx Interactive.

Andrea Bizzotto received his BS and MS degrees in computer engineering from the University of Padua, Italy. After completing college he joined Imagination Technologies, where he developed a range of 3D demos for Imagination's POWERVR Insider ecosystem and published an article in *GPU Pro*. His research interests include 3D graphics, computer vision, image processing, algorithm theory, and software design. More info at www.bizzotto.biz.

Samuel Boivin is a research scientist at INRIA in leave of absence. He is now the head of Research and Development at SolidAnim, a company specializing in Visual Effects for movies and video games. He earned a PhD in computer graphics in 2001 from Ecole Polytechnique in Palaiseau (France). He has published several papers about computer graphics in many conferences, including SIGGRAPH. His research topics are photo-realistic real-time rendering, real-time augmented reality, fluid dynamics, and inverse techniques for acquiring material properties (photometric, mechanical) from videos.

Xavier Bonaventura is currently a PhD student at the Graphics & Imaging Laboratory of University of Girona, researching on viewpoint selection. He developed his master's thesis on hardware tessellation at Budapest University of Technology and Economics, within the Erasmus program.

George Borshukov is CTO of embodee, a technology company that helps anyone find, select, and personalize apparel using a fun yet useful process. He holds an MS from the University of California, Berkeley, where he was one of the creators of *The Campanile Movie* and real-time demo (1997). He was technical designer for the "bullet time" sequences in *The Matrix* (1999) and received an Academy Scientific and Technical

Achievement Award for the image-based rendering technology used in the film. Borshukov led the development of photoreal digital actors for *The Matrix* sequels (2003) and received a Visual Effects Society Award for the design and application of the universal capture system in those films. He is also a co-inventor of the UV pelting approach for parameterization and seamless texturing of polygonal or subdivision surfaces. He joined Electronic Arts in 2004 to focus on setting a new standard for facial capture, animation, and rendering in next-generation interactive entertainment. As director of creative R&D for EA SPORTS he led a team focused on expanding the label's brand and its interactive entertainment offerings through innovation in hi-fidelity graphics and new forms of interactivity, including 3D camera devices.

Ken Catterall graduated from the University of Toronto in 2005 as a specialist in software engineering. He is currently a leading engineer for Imagination Technologies' business development team, where he has been working since 2006 developing and supporting Imagination's key graphics demos. He has previously contributed to *ShaderX*[6], *ShaderX*[7], and *GPU Pro*.

Fabrice Colin earned a PhD in mathematics from the University of Sherbrooke (Canada) in 2002, under the supervision of Dr. Kaczynski (University of Sherbrooke) and Dr. Willem (Universit catholique de Louvain), with a thesis in the field of partial differential equations (PDE). In 2005 he was hired by the Mathematics and Computer Science Department at Laurentian University, where he is currently a professor. His primary research interests are the variational and topological methods in PDE. But besides his theoretical work, he started collaborating in 2003 with Professor Egli, from the Department of Computer Science (University of Sherbrooke), on the numerical simulations of PDE related, for instance, to fluid dynamics or computer graphics.

Daniel Collin is a senior software engineer at EA DICE, where he for the past 5 years he spent most of his time doing CPU and memory optimizations, tricky low-level debugging, and implementing various culling systems. He is passionate about making code faster and simpler in a field that moves rapidly.

Joe Davis graduated from the University of Hull in 2009 with a MEng degree in computer science, where his studies focused on real-time graphics and physics for games. Joe currently works as a developer technology engineer for the Imagination Technologies POWERVR Graphics SDK, where his time is spent developing utilities, demos, and tutorials as well as helping developers optimize their graphics applications for POWERVR-based platforms.

Jose I. Echevarria received his MS degree in computer science from the Universidad de Zaragoza, Spain, where he is currently doing research in computer graphics. His research fields range from real-time to off-line rendering, including appearance acquisition techniques.

Richard Egli has been a professor in the Department of Computer Science at University of Sherbrooke since 2000. He received his BSc degree and his MSc degrees in computer science at the University of Sherbrooke (Québec, Canada). He received his PhD in computer science from the University of Montréal (Québec, Canada) in 2000. He is the

chair of the MOIVRE research center (Modélisation en Imagerie, Vision et Réseaux de neurones). His research interests include computer graphics, physical simulations, and artificial life.

Holger Gruen ventured into 3D real-time graphics right after his university graduation, writing fast software rasterizers in 1993. Since then he has held research and also development positions in the middleware, games, and the simulation industries. He addressed himself to doing developer relations in 2005 and now works for AMD's product group. Holger, his wife, and his four kids live in Germany close to Munich and near the Alps.

Martin Guay completed a BSc in mathematics in 2007 and then Martin Guay joined Cyanide's Montreal-based studio, where he worked on the development of several game titles as a graphics programmer. In 2010 he joined the MOIVRE research centre, Université de Sherbrooke, as a graduate student in computer science, where he effectuated research in the fields of computational physics and physically based animation of fluids.

Diego Gutierrez is a tenured associate professor at the Universidad de Zaragoza, where he got his PhD in computer graphics in 2005. He now leads his group's research on graphics, perception, and computational photography. He is an associate editor of three journals, has chaired and organized several conferences, and has served on numerous committees, including the SIGGRAPH and Eurographics conferences.

Ralf Habel is a postdoctoral researcher at the Institute of Computer Graphics and Algorithms of the Vienna University of Technology, where he received his PhD in 2009. He studied theoretical physics and computer graphics at the University of Stuttgart and the Georgia Institute of Technology. His current research interests are real-time rendering, precomputed lighting and transfer, and computational photography.

Benjamín Hernández earned a PhD in Computer Science from Instituto Tecnológico y de Estudios Superiores Monterrey, Campus Estado de México, under the supervision of Isaac Rudomin. He recently joined the faculty at Instituto Tecnológico y de Estudios Superiores Monterrey, Campus Ciudad de México. He has published several papers in the field of computer animation, crowd behavior, GPGPU programming, and virtual reality for art applications. His research interests include animation of virtual characters, procedural generation of crowds, rendering of complex scenes, and interaction design for mobile devices.

Stephen Hill is a 3D technical lead at Ubisoft Montreal, where for the past 6 years he has been single-mindedly focused on graphics R&D for the Splinter Cell series (Conviction and Chaos Theory). For him, real-time rendering in games is the perfect storm of artistic and technical creativity, low-level optimization, and pure showing off, all within a fast-paced field.

Jorge Jimenez is a real-time graphics researcher at the Universidad de Zaragoza, in Spain, where he received his BSc and MSc degrees, and where he is pursuing a PhD in real-time graphics. His passion for graphics started after watching old school demos in his brother's Amiga A1000. His interests include real-time photorealistic rendering,

special effects, and squeezing rendering algorithms to be practical in game environments. He has numerous contributions in books and journals, including *Transactions on Graphics*, where his skin renderings made the front cover of the SIGGRAPH Asia 2010 issue. He loves challenges, playing games, working out in the gym, and more than anything, breaking in the street.

Anton Kaplanyan is a Lead Researcher at Crytek. During the development of Crysis 2 and CryEngine 3 he was responsible for research on graphics and porting of CryEngine 2 to the current generation of consoles. Currently he is busy working on the next iteration of the engine to keep pushing both DX11 and next-gen console technology. Additionally he has been working on his PhD thesis within Karlsruhe University since 2009. Prior to joining Crytek he received his MS in computer science at Moscow University of Electronic Engineering, Russia, in early 2007.

Andrew Lauritzen is a software engineer on the Advanced Rendering Technology team at Intel. He received his M.Math in computer science from the University of Waterloo in 2008, where his research was focused on variance shadow maps and other shadow filtering algorithms. His current research interests include lighting and shadowing algorithms, deferred rendering, parallel programming languages, and graphics hardware architectures.

Olivier Le Maître is a member of the research staff at the French National Center for Research (CNRS), working in the Department of Mechanics and Energetics of the LIMSI lab. After receiving a PhD in computational fluid dynamics in 1998 he joined the University of Evry, where he taught scientific computing, numerical methods, and fluid mechanics. He joined CNRS in 2007 as a full-time researcher, where he directs research in complex flow simulations, uncertainty quantification techniques, and stochastic models.

Aaron Lefohn is a senior graphics architect at Intel, where he is a research lead in the advanced rendering technology team, creating new interactive rendering algorithms, pipelines, and programming models. Aaron previously led Intel's involvement in OpenCL, where he contributed significantly to OpenCL's heterogeneous parallel coordination API. Before joining Intel he designed parallel programming models for rendering on Sony PlayStation3 at the startup Neoptica. Aaron spent three years at Pixar working on interactive rendering tools for artists and GPU acceleration of RenderMan. He received his PhD in computer science from UC Davis in 2006 and holds MS and BA degrees from the University of Utah and Whitman College.

Belen Masia received her MS degree in computer science and systems engineering from the Universidad de Zaragoza, Spain, where she is currently a PhD student. Her research interests lie somewhere between the fields of computer graphics and computer vision, currently focusing on image processing and computational photography. Her work has already been published in several venues, including *Transactions on Graphics*.

Oliver Mattausch is a researcher at the Institute of Computer Graphics and Algorithms of the Vienna University of Technology. From the same university he received an MSc

in 2004 and his PhD in 2010. He coauthored several papers in the fields of real-time rendering and scientific visualization. His current research interests are real-time rendering, visibility computations, shadow algorithms, and real-time global illumination.

Fernando Navarro works as lead technical artist for Lionhead Studios (Microsoft Games Studios). Prior to that he directed the R&D departments of different production houses. His experience covers vfx, feature films, commercials, and games. He has a MSc from the Universidad de Zaragoza and is currently pursuing a PhD in computer science, focused on advanced rendering algorithms.

Aleksander Netzel is a masters degree student at the University of Wrocław. He is currently employed as a graphic programmer at Techland, where he adapts the latest cutting-edge graphic techniques for the current generation of gaming platforms. His goal is to develop new solutions for real-time graphics problems as part of a research team.

Anders Nilsson is a software engineer at Illuminate Labs, working on precomputed lighting for games. He is an obsessive graphics coder with a MSc in engineering mathematics from Lund University.

Sebastien Noury is a PhD student in computer science at Paris-Sud University in Orsay, France, where he received his MSc in software engineering in 2009. He also studied game development in Montreal and interned at independent game studios in Paris and Singapore before joining the VENISE team of the CNRS/LIMSI lab to work on real-time fluid dynamics for virtual reality. His research interests include GPU acceleration of dynamic simulations, real-time rendering, and human-computer interaction.

Christopher Oat is a graphics lead at Rockstar Games, where he works on real-time rendering techniques used in Rockstar's latest titles. Previously he was the demo team lead for AMD's Game Computing Applications group. Christopher has published his work in various books and journals and has presented at graphics and game developer conferences worldwide. Many of the projects that he has worked on can be found on his website: www.chrisoat.com.

Eric Penner is a rendering engineer at Electronic Arts and a research associate at the Hotchkiss Brain Institute Imaging Informatics lab at the University of Calgary. He holds a MSc degree from the University of Calgary, Alberta, where he worked on GPU-accelerated medical volume rendering algorithms. Eric's MSc work is being commercialized by Calgary Scientific Inc. At Electronic Arts Eric has filled the roles of lead programmer on a Creative R&D team focused on cutting edge rendering and new forms of controller free interaction, as well as rendering engineer on the NHL series of games. Prior to working at Electronic Arts, Eric was a rendering engineer on the Advanced Technology group at Radical Entertainment and worked on the popular games *Prototype* and *Scarface*.

Matt Pharr is a principal engineer at Intel and the lead graphics architect in the Advanced Rendering Technology group. He previously co-founded Neoptica, worked in the Software Architecture group at NVIDIA, co-founded Exluna, worked in Pixar's

Rendering R&D group, and received his PhD from the Stanford Graphics Lab. With Greg Humphreys, he wrote the textbook *Physically Based Rendering: From Theory to Implementation*. He was also the editor of *GPU Gems 2*.

Ferenc Pintér is an engine programmer at Digital Reality's Technology Team. After working on network congestion control systems for Ericsson, he joined Eidos Hungary to create *Battlestations: Pacific* for PC and X360. His passion for outdoor rendering, procedurally aided content creation, and artist-friendly graphics tools has only been getting more fueled ever since.

Peter Quayle graduated from the University of Brighton with a degree in computer science. His interest in real-time computer graphics originates from a fascination with the demoscene. Peter currently works at Imagination Technologies as a business development engineer.

Donald Revie graduated from the University of Abertay with a BSc (Hons) in computer games technology before joining Cohort Studios in late 2006. He has worked on Cohort's Praetorian Tech platform since its inception, designing and implementing much of its renderer and core scene representation. He has also had the opportunity to develop his interest in graphics programming through working on shaders and techniques across various projects. He has continued to help develop Praetorian and Cohort's shader library across four released titles and numerous internal prototypes. He finds writing about himself in the third person incredibly unsettling.

Philip Rideout works at Medical Simulation Corporation in Denver, where he develops novel rendering techniques for fluoroscopy and ultrasound simulation. Philip has written a book on 3D programming for the iPhone and maintains a blog for graphics tricks at http://prideout.net. In his spare time he can be found watching *Doctor Who* with Sreya, Pragati, and Arnav.

Pawel Rohleder claims he has been interested in the computer graphics and game industry since he was born. He is keen on knowing how all things (algorithms) work, then trying to improve them and developing new solutions. He started programming games professionally in 2002; he has been 3D graphics programmer at Techland's ChromeEngine team for three years and since 2009 has been working as a Build Process Management Lead / R&D manager. He is also a PhD student in computer graphics (at Wroclaw University of Technology, since 2004).

Isaac Rudomin earned a PhD in computer science from the University of Pennsylvania in 1990, with a dissertation "Simulating Cloth using a Mixed Geometrical-Physical Method," under the guidance of Norman I. Badler. He joined the faculty at Instituto Tecnológico y de Estudios Superiores Monterrey, Campus Estado de México, in 1990 and from that date on he has been active in teaching and research. He is interested in many areas of computer graphics and has published a number of papers. Lately his research has an emphasis in human and crowd modeling, simulation, and rendering.

Marco Salvi is a senior graphics engineer in the Advanced Rendering Technology group at Intel, where he focuses his research on new interactive rendering algorithms and sw/hw graphics architectures. Marco previously worked for Ninja Theory and LucasArts as a graphics engineer on multi-platform and PS3-exclusive games where he was responsible for architecting renderers, developing new rendering techniques and performing low-level optimizations. Marco received his MSc in physics from the University of Bologna in 2001.

Daniel Scherzer is a researcher at the Ludwig Boltzmann Institute for Archaeological Prospection and Virtual Archaeology. He is also lecturer at the Vienna University of Technology and the FH Hagenberg. He was assistant professor at the Institute of Computer Graphics and Algorithms of the Vienna University of Technology, where he received an MSc in 2005, an MSocEcSc in 2008, and a PhD in 2009. His current research interests include temporal coherence methods, shadow algorithms, modeling, and level-of-detail approaches for real-time rendering; he has authored and coauthored several papers in these fields.

Wojciech Sterna has been interested in computer graphics and games development since 2004. He is keen on implementing graphics algorithms (shadows especially) and never lets go 'til he understand all the theoretical basics that lie behind. Wojtek is currently finishing a remake of *Grand Theft Auto 2* called *Greedy Car Thieves* (http://gct-game.net) and hopes that players will love that game as much as he does.

Michael Schwärzler is a PhD student, researcher, and project manager in the field of real-time rendering at the VRVis Research Center in Vienna, Austria. In 2009, he received his master's degree in computer graphics and digital image processing at the Vienna University of Technology, and his master's degree in computer science and management at the University of Vienna. His current research efforts are concentrated on GPU lighting simulations, real-time shadow algorithms, image-based reconstruction, and semantics-based 3D modeling techniques.

Nicolas Thibieroz has more than 12 years of experience working in developer relations for graphics hardware companies. He taught himself programming from an early age as a result of his fascination with the first wave of "real-time" 3D games such as Ultima Underworld. After living in Paris for 22 years, Nicolas decided to pursue his studies in England, where he earned a bachelor of electronic engineering in 1996. Not put off by the English weather, Nicolas chose to stay and joined PowerVR Technologies to eventually lead the developer relations group, supporting game developers on a variety of platforms and contributing to SDK content. He then transitioned to ATI Technologies and AMD Corporation, where his current role involves helping developers optimize the performance of their games and educating them about the advanced features found in cutting-edge graphics hardware.

Kiril Vidimce is a senior software architect and researcher at Intel's Advanced Rendering Technologies group. His research and development interests are in the area of real time rendering, cinematic lighting, physically-based camera models, and computational photography. Previously he spent eight years at Pixar as a member of the R&D group

working on the in-house modeling, animation, lighting, and rendering tools, with a brief stint as a Lighting TD on Pixar's feature film, *Cars*. In his previous (academic) life he did research in the area of multiresolution modeling and remeshing. His research work has been published at SIGGRAPH, EGSR, IEEE Visualization, IEEE CG&A, and Graphics Interfaces.

Marco Weber received a MS degree in computer science from the Friedrich-Alexander University of Erlangen-Nuremberg with a focus on computer graphics and pattern recognition. Currently, he is a developer technology engineer at Imagination Technologies doing research, implementing demos for new and upcoming technologies, and helping fellow graphics programmers optimize their algorithms and engines for POWERVR-based mobile platforms.

Michael Wimmer is an associate professor at the Institute of Computer Graphics and Algorithms of the Vienna University of Technology, where he received a MSc in 1997 and a PhD in 2001. His current research interests are real-time rendering, computer games, real-time visualization of urban environments, point-based rendering, and procedural modeling. He has coauthored many papers in these fields, and was papers co-chair of EGSR 2008.