# Advances in Real-Time Rendering in 3D Graphics and Games

## SIGGRAPH 2008 Course
## Part I and II

**August 11, 2008**


**Course Organizer and Notes Editor: Natalya Tatarchuk, AMD**

**Lecturers:**

**Michael Boulton, Rare**
**Hao Chen, Bungie**
**Dominic Filion, Blizzard Entertainment**
**Xinguo Liu, Microsoft Research Asia / Zhejiang University**
**Martin Mittring, Crytek GmbH**
**Rob McNaughton, Blizzard Entertainment**
**Christopher Oat, AMD**
**Natalya Tatarchuk, AMD**

# About This Course

Advances in real-time graphics research and the increasing power of mainstream GPUs have resulted in an explosion of innovative algorithms suitable for rendering complex virtual worlds at interactive rates. Every year the latest video games display a vast variety of sophisticated algorithms resulting in ground-breaking 3D graphics that push the visual boundaries of interactive experience.

This course will cover a number of topics ranging from the best practices and techniques prevalent in current state-of-the-art rendering in many award-winning games all the way up to innovative 3D rendering research that will be found in the games of tomorrow. This will include examples from recently games from Crytek, Rare, Bungie as well as upcoming titles from Blizzard Entertainment, and graphics research from AMD's Game Computing Applications Group.

## Prerequisites

*This course assumes working knowledge of a modern real-time graphics API like OpenGL or Direct3D, as well as a solid basis in commonly used graphics algorithms. The participants are also assumed to be familiar with the concepts of programmable shading and shading languages.*

**Intended audience**: *Technical practitioners and developers of graphics engines for visualization, games or effects rendering interested in interactive rendering. Presented techniques are applicable to the real-time and offline domains. The attendees will come away with a number of highly optimized algorithms in various areas of real-time rendering.*

## Topics

- Lighting and Material of Halo 3
- Advanced Virtual Texture Topics
- March of the Froblins: Rendering massive crowds of intelligent and detailed creatures on GPU
- Using Wavelets with Current and Future Hardware
- Rendering Techniques from StarCraft II

## Suggested Reading


- Real-Time Rendering by Tomas Akenine-Möller, Eric Haines, and Naty Hoffman, A.K. Peters, Ltd.; 3rd edition, 2008
- Advanced Global Illumination by Philip Dutre, Phillip Bekaert, Kavita Bala, A.K. Peters, Ltd.; 1st edition, 2003
- Radiosity and Global Illumination by François X. Sillion, Claude Puech; Morgan Kaufmann, 1994.
- Physically Based Rendering : From Theory to Implementation by Matt Pharr, Greg Humphreys; Morgan Kaufmann; Book and CD-ROM edition (August 4, 2004)
- The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics, Steve Upstill, Addison Wesley, 1990.
- Advanced RenderMan: Creating CGI for Motion Pictures, Tony Apodaca & Larry Gritz, Morgan-Kaufman 1999.
- Texturing and Modeling, A Procedural Approach Second Edition, Ebert, Musgrave, Peachey, Perlin, Worley, Academic Press Professional, 1998.
- ShaderX6: Advanced Rendering, by Wolfgang Engel (Editor), Charles River Media, 1st edition, (February 2008)
- ShaderX5: Advanced Rendering Techniques, by Wolfgang Engel (Editor), Charles River Media, 1st edition (December 2006)
- ShaderX4: Advanced Rendering Techniques, by Wolfgang Engel (Editor), Charles River Media, 1st edition (November 2005)
- ShaderX3: Advanced Rendering with DirectX and OpenGL, by Wolfgang Engel (Editor), Charles River Media, 1st edition (November 2004)
- ShaderX2: Introductions and Tutorials with DirectX 9.0, by Wolfgang Engel (Editor), Wordware Publishing, Inc.; Book and CD-ROM edition (November 2003), now free online: http://www.gamedev.net/reference/programming/features/shaderx2/Introductions_and_Tutorials_with_DirectX_9.pdf
- ShaderX2 : Shader Programming Tips and Tricks with DirectX 9.0, by Wolfgang Engel (Editor), Wordware Publishing, Inc.; Book and CD-ROM edition (November 2003), now free online: http://www.gamedev.net/reference/programming/features/shaderx2/Tips_and_Tricks_with_DirectX_9.pdf

# Lecturers

**Natalya Tatarchuk, Graphics SW Architect, AMD**

Natalya is a graphics software architect and a project lead in the Game Computing Application Group at AMD Graphics Products Group (Office of the CTO). There she pushes parallel computing boundaries investigating innovative real-time graphics techniques. In the past she has been the lead of ATI's demo team creating the state-of-the-art interactive renderings and has been the lead for the tools group at ATI Research. Prior to that Natalya worked on 3D modeling software, and scientific and financial visualization, among other projects. She has published papers and articles in various computer graphics conferences and technical book series, and has presented her work at graphics and game developer conferences worldwide.

**Christopher Oat, MTS, AMD**

Christopher Oat is a member of AMD's Game Computing Applications Group (Office of the CTO) where he is a technical project lead working on state-of-the art demos. In this role, he focuses on the development of cutting-edge rendering techniques for the latest graphics platforms. Christopher has published his work in various books and journals and has presented his work at graphics and game developer conferences around the world.

**Dominic Filion, Senior Software Engineer, Blizzard Entertainment**

Dominic is currently a senior software engineer at Blizzard Entertainment, where he has been hard at work on the upcoming Starcraft II for the past few years. He has worked for close to a decade in the games industry, acting as technical director or principal architect on three different commercial 3D engines at several game companies prior. On the rare moments where he is not obsessing about improving Starcraft II's graphics, Dominic would enjoy feedback on the material presented here, so feel free to drop him a note!

**Rob McNaughton, 3D Animator and Digital Effects artist, Blizzard Entertainment**

Rob McNaughton is a Southern California native bent on playing games for a living. That works out since he has been employed at Blizzard Entertainment for over 12 years. Rob currently is Lead Technical Artist for Blizzard's Team 1, and has worked on the following games for them: StarCraft II (When it is ready), World of Warcraft: The Burning Crusade (2007) World of Warcraft (2004), WarCraft III: The Frozen Throne (2003), Warcraft III: Reign of Chaos (2002), StarCraft (1998), StarCraft: Brood War (1998), Diablo (1996).

Rob is primarily a 3D Animator and Digital Effects artist but has done his time at many art tasks including pencil sketching and digital painting. Digital speed painting has become a favorite new work medium brought on with help of conceptart.org.

**Hao Chen, Graphics Architect, Bungie Studio**

Hao Chen is the graphics architect and one of the engineering lead for Bungie Studio, where he currently leads the research and development of Bungie's next generation graphics engine. He was the graphics engineering lead of Halo3. Prior to that, Hao has worked on numerous game titles for Microsoft and Bungie on the Xbox and PC platforms, including Outwars, AMPED1, AMPED2, and Halo2.

**Xinguo Liu, Professor, Zhejiang University**

Xinguo Liu is a professor of the Computer Science School at Zhejiang University. His research interests include geometry processing, appearance modeling, real-time rendering, and deformable objects. He received a B.Sc. in 1995 and a Ph.D. in 2001 from Zhejiang University. He was a researcher at the Internet Graphics Group of Microsoft Research Asia from 2001 to 2006, and he was a visiting researcher at CMU Graphics Lab in 2007.

**Michael Boulton, Senior Software EngineerRare/MGS**

Michael has worked at Rare/MGS for over five years, and is currently a senior software engineer. He wrote the graphics engine for VIVA PINATA on the Xbox360, and has given previous presentations at both GDC and ACM SIGGRAPH. Currently, he works in the shared technology department at Rare, developing technology for current and future hardware.

**Martin Mittring, Lead Graphics Programmer, Crytek GmbH**

Martin is a software engineer and member of the R&D staff at Crytek. Martin started his first experiments early with text-based computers, which led to a passion for computer and graphics in particular. He studied computer science and worked in one other German games company before he joined Crytek. During the development of Far Cry he was working on improving the Polybump™ tools and was became lead network programmer for that game. His passion for graphics brought him back to former path and so he became lead graphics programmer in R&D. Currently he is busy working on the next iteration of the engine to keep pushing future PC and next-gen console technology.

# Contents

# Preface

Welcome to the *Advances in Real-Time Rendering in 3D Graphics and Games* course at SIGGRAPH 2008. We're excited to bring you the third installment of our successful course, with lectures from top game developers as well as industry researchers. We have included both *3D Graphics* and *Games* in our course title in order to emphasize the incredible relationship that has rapidly grown between the graphics research and the game development communities in the recent years. Long gone are the days when interactive rendering was synonymous with gross approximations and careless assumptions, or simplistic visual rendering. With the amazing evolution of the processing power of consumer-grade GPUs, the gap between offline and real-time rendering is rapidly shrinking. Now the question becomes less of "how can I simplify what I'm doing?", but, with over a teraflop of compute on a single GPU, we can now ask the question of "What amazing effect do I want to do now?" Truly, the frontiers of interactive rendering research have expanded and the real-time domain is now at the forefront of state-of-the-art graphics research.

As researchers, we focus on pushing the boundaries with innovative computer graphics theories and algorithms. As game developers, we bend the existing software APIs such as DirectX and OpenGL and the available hardware to perform our whims at interactive rates. And as graphics enthusiasts we all strive to produce stunning images that come alive on our screens. It is this synergy between researchers and game developers that is driving the frontiers of interactive rendering to create truly rich, immersive environments. There is no greater satisfaction for developers than to share the lessons learned and to see our technologies used in ways we never imagined.

This is the third time this course is presented at SIGGRAPH and we hope that you enjoy the new material presented this year and come away with a new understanding of what is possible (without sacrificing interactivity!). We hope that we will inspire you to help drive real-time rendering research and games!

Natalya Tatarchuk, AMD
August, 2008

# Chapter 1

# Lighting and Material of *Halo 3*

Hao Chen[1]
Xinguo Liu[2]

***Figure 1.*** *A still frame from* Halo 3 *showing the main character rendered with complex materials under global illumination.*

[1] haochen@bungie.com
[2] xgliu@cad.zju.edu.cn

## 1.1 Introduction

Lighting and material are very important aspects of the visual appearances of games and they present some of the hardest challenges in real time graphics today. For *Halo* and indeed many other games, keeping the players immersed in the virtual environment for long periods of time is a top priority of the graphics system, and good quality lighting and realistic materials are the fundamental building blocks for achieving the level of realism necessary to accomplish this goal.

***Figure 2.*** *A still frame from* Halo 3 *in diffuse only mode that demonstrates the effect of global illumination.*

Except in special cases such as in a highly stylized environment, a good lighting scheme must be able to capture some form of global illumination, as can be seen here in Figure 2. Rendering global illumination involves solving the rendering equation first proposed in [IGC86][KAJIYA86]. If we assume that a surface is not self-emissive, then the rendering equation can be written as:

$$I(V) = \int_{\Omega} f(V, L)\cos(\theta)\ell(\omega)d\omega \quad , \qquad (1)$$

The equation states that the total reflected light energy $I$ from a surface point along a viewing direction $V$ is the product of the BRDF $f$, the incoming light $\ell$, and the cosine of the angle $\theta$ between the view and the light direction (*V* and *L*, respectively), integrated over the hemisphere defined by the surface normal (See Table 1 for a list of symbols and their definitions used throughout the chapter).

Solving the equation directly is generally not feasible for a real time game, except for special cases such as when the lighting environment is made of only a small number of point light sources and when we only consider direct illumination.

***Figure 3.*** *An outdoor scene made of different materials.*

Rendering complex materials under global illumination presents further challenges. A typical scene in *Halo 3* is made of many different materials, from dull concrete to shiny metal, as seen in Figure 3. Although simple BRDF models such as Phong have been used extensively in real time games, they are only trivial to evaluate interactively if the light source are point lights. For area light sources like the sky light in Figure 3, rendering the BRDF involves evaluating an expensive integral, as in Equation 1. The Phong model also does not capture important physical properties such as the Fresnel effect, which can contribute significantly to the realism of many materials.

For *Halo 3*, our goal is to be able to handle arbitrary light sources and to capture important global illumination effects, such as the soft lighting bounced from the floor towards the tree trunk as seen in Figure 2. We also want to render a large variety of materials under global and environment lighting, using a more physically accurate BRDF model.

In this chapter we describe the core lighting and material models used in *Halo 3*. We first describe the Cook Torrance BRDF model. Then we show how Equation 1 can be factored into separate components, which can each be rendered in real time using different techniques, and how the parts are combined together in the end for final shading. The shader code and further details are listed in the Appendix.

| | |
|---|---|
| $V$ | unit vector in the viewer direction |
| $L$ | unit vector in the lighting direction |
| $H$ | halfway vector: unit vector in $V + L$ |
| $m$ | material roughness parameter |
| $k_d$ | diffuse reflectance coefficient |
| $k_s$ | specular reflectance coefficient |
| $R_d$ | diffuse reflectance |
| $R_m$ | roughness function |
| $F_0$ | Fresnel value at normal incidence angle |
| $F$ | Fresnel term |
| $G$ | geometrical attenuation factor in Cook-Torrance model |
| $D$ | micro-facet distribution function in Cook-Torrance model |

***Table 1.*** *Symbols used in this work*

## 1.2 The Cook Torrance BRDF

The Cook-Torrance BRDF, first introduced in [COOKTORRANCE81], is based on the micro-facet theory, and is found to be more accurate than Blinn-Phong [BLINN77] model commonly used in games [NDM05]. We pick this model because it offers a good trade-off between BRDF expressiveness and computational complexity. Our methods can be extended to other models (See Appendix for an implementation of the Phong model). The Cook Torrance reflectance model is as follows:

$$f(V,L) = k_d R_d + k_s F R_m(V,L), \qquad (2)$$

where the first term is the diffuse component, which is dependent on the incoming light only, and the second term is the specular component, which is dependent on both the light and the viewing directions. Plugging the BRDF into the rendering equation in (1), we have:

$$I(V) = k_d R_d \int \cos(\theta)\ell(\omega)d\omega + k_s \int F R_m(V,L)\cos(\theta)\ell(\omega)d\omega \qquad (3)$$

Both the diffuse and specular components in Equation 3 involve an integral that is expensive to compute directly in real time. However, if the light sources are point lights only, the integral becomes a sum of ( $N \cdot L$ ) terms for diffuse illumination. The specular reflectance can be evaluated in a similar fashion by evaluating the specular lobe directly for each point light and then summing. Games have used these simple forms of lighting and material models extensively, some to great effect. However, the restriction of point light sources (and direct illumination only) is one of the main reasons that many games

look unrealistic. To handle arbitrary light sources we need to find a way to evaluate the integrals in equation 3 efficiently, and for that we turn to spherical harmonics.

## 1.3 Spherical Harmonics Light Maps and Diffuse Reflectance

Spherical harmonics (SH) basis over the sphere is analogous to the Fourier basis over the line or the circle. It can be shown that for Lambertian surfaces, a small number of spherical harmonics terms are sufficient to approximate the original lighting to high accuracy [BRASRIJACOBS03]. The *Spherical Harmonics Irradiance Environment Maps*, introduced in [RAMAMOORTHIHANRAHAN01], encode the irradiance distribution function as a vector of SH coefficients. The un-shadowed diffuse transfer can then be computed in a shader using a quadratic polynomial approximation. For shadowed transfer and inter-reflections, the *Pre-computed Radiance Transfer* (PRT) method [SKS02] pre-computes the transfer function as a SH vector, which is then combined with incoming lighting using a SH dot product. Glossy materials are also possible in the SH representation [KSS02][RAMAMOORTHIHANRAHAN02][SHHS03], although none of the previous methods can meet the stringent performance and storage requirements of a real time game. We chose spherical harmonics basis for our lighting and material methods as well for the following reasons: SH is suitable for approximating smooth changing signals using a small number of coefficients which makes it ideal for lighting, transfer and material purposes; SH can be rotated easily in a shader; Finally, there are a number of algorithms such as PRT that are compatible with the SH representation which we use in our game.

The incoming lighting incident upon a point can be projected into spherical harmonics as follows:

$$\lambda_i = \int_{\Omega} \ell(\omega) Y_i(\omega) d\omega \qquad (4)$$

where $Y_i$ are the spherical harmonics: $Y_{00}$, $Y_{1,-1}$, $Y_{1,0}$, $Y_{1,-1}$, $Y_{2,-2}$, $Y_{2,-1}$, $Y_{2,0}$, , $Y_{2,1}$, $Y_{22}$. Using SH basis, the diffuse reflectance in Equation 3 is simply:

$$I_d(V) = k_d R_d \int_{\Omega} \cos(\theta) \ell(\omega) d\omega = k_d R_d \sum_{i=0,2,6} \lambda_i A_i \qquad (5)$$

Here $R_d$ is the Lambertian BRDF which is a constant, and $A_i$ is the projection of the cosine lobe in SH. Since the cosine lobe is radially symmetric around the normal, the coefficients of its SH projection are non-zero only for the $i$ = 0, 2, 6, … basis. The first three terms are given as (from [RAMAMOORTHIHANRAHAN01B]):

$$A_0 = \sqrt{\pi / 4}, \; A_1 = \sqrt{\pi / 3}, \; A_2 = \sqrt{5\pi / 64}$$

Notice equation 5 assumes that the incoming light is rotated into the local coordinate frame of the surface point.

Equation 4 encodes the incident radiance at a single point. This is insufficient for our purpose of lighting the whole scene since the incident radiance is spatially varying from point to point. A commonly used strategy to approximate such spatially varying functions is to sample the function at discrete sampling points, and then interpolate between the samples everywhere. There are several possible strategies we can use.

One such strategy is to spatially divide the scene into cells using a regular 3D grid, then store one or more samples per cell. This is the original scheme used in [GSHG98].

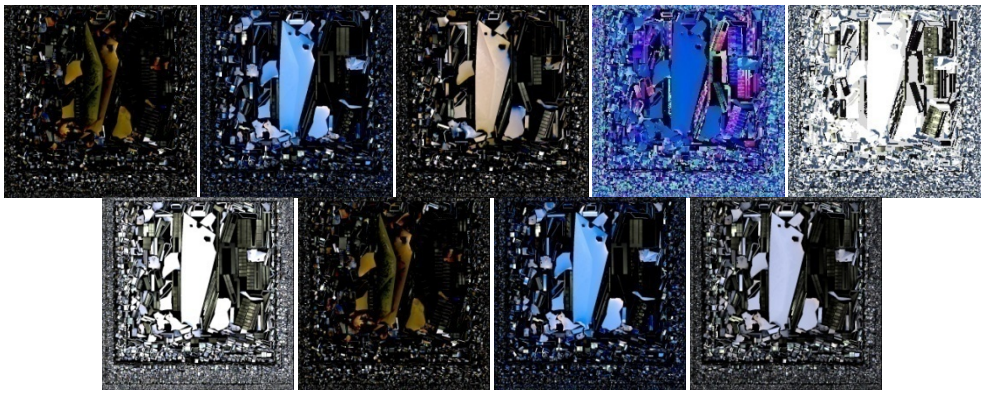

***Figure 4.*** *Harmonics lightmap textures using quadratic SH.*

In the ATI *Ruby: Dangerous Curves* demo [OAT05], an adaptive subdivision method is proposed to reduce the number of samples needed while preserving important details.

***Figure 5.*** *Hard shadow edges and bump mapping are hard problems for schemes based on discrete irradiance volumes.*

The ATI demo also uses spherical harmonics gradients to improve the interpolation between samples. The spatial subdivision scheme is well suited for rendering small dynamic objects, since the whole object can be rendered from one or a few lighting samples, obtained by interpolating between the samples closest to the object's location. For large static environment geometry however, the spatial subdivision scheme has several hard-to-overcome problems as shown here in Figure 5. First, rendering bump maps from such a scheme would require associating each pixel with one or more samples closest to the pixel location, and this is non-trivial and expensive to do in real time. Furthermore, dense samples must be stored along a shadow boundary, in order to preserve the appearance of sharp shadows, and this would require very final levels of subdivisions.

In *Halo 3*, we represent the lighting as spherical harmonics light maps, as shown in Figure 4. They can be viewed as analogous to traditional light maps, but instead of

storing the exit radiance as a single color, each pixel stores the incident radiance at the surface point, encoded in SH as in Equation 4. A similar representation is used in [GOODTAYLOR05] as an optimization for photon mapping.

SH light maps are textures parameterized over the surface which can be mapped to the geometry in real time. These textures can be generated using any offline global illumination solver. We use a distributed photon mapping algorithm running on a multi-node rendering farm. See [CHEN08][VILLEGASSEAN08] for further details.

Rendering from SH light maps is straightforward. A key benefit of the SH light maps is that bump maps can now be rendered directly. We now show a couple of examples of the diffuse reflectance of an environment rendered directly from the SH light map representation.

***Figure 6.*** *An Outdoor scene from* Halo 3. *The sun and sky models are from [PSS99], they are the only light source in the scene. Global illumination is the key factor to the realism of this scene, so is rendering everything from a consistent lighting representation.*

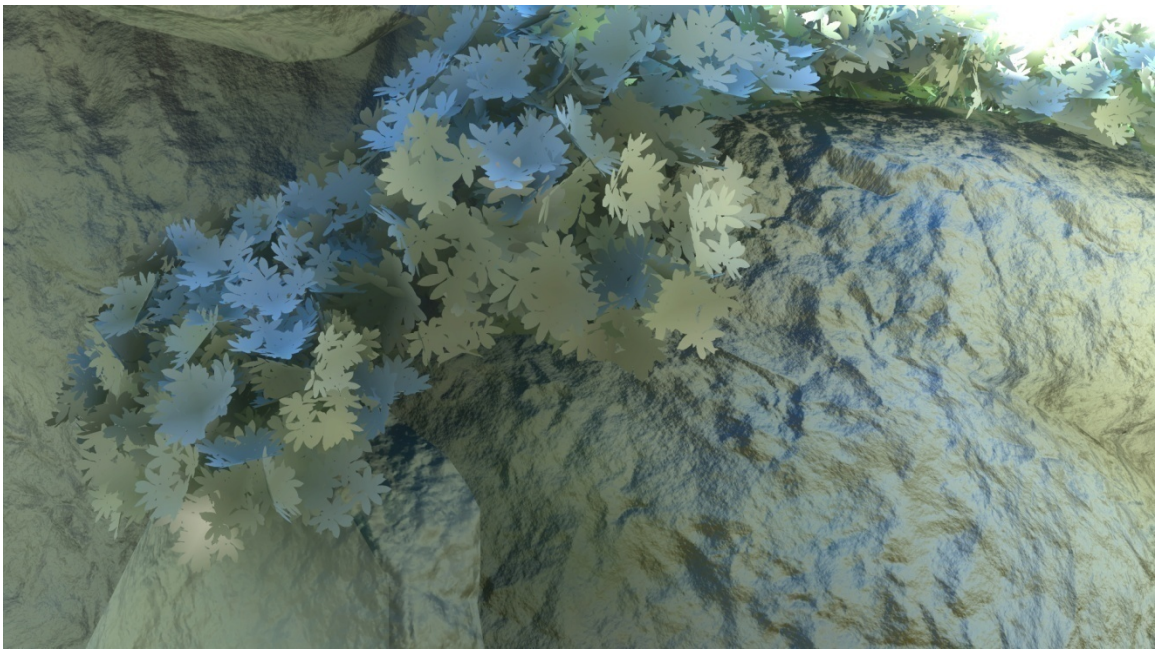

***Figure 7.*** *Close-up view of a bump mapped surface. Notice the subtle shading of the bumped detail from reflected lighting below and the large, bluish sky light. This is very difficult to achieve with point lights, unless a large number of them are used.*

## 1.4 Specular Reflectance

Rendering glossy material under global and environment lighting is hard for real time games. There are several reasons for this. First, the specular reflectance is a view dependent quantity which makes the specular part in Equation 3 much more difficult to evaluate than the diffuse part. Second, a glossy material typically has both high and low frequency parts and everything in-between, but low order spherical harmonics is effectively a low pass filtered signal of the original, and thus ill suited for capturing high frequency glossy appearance.

Our method is to breakdown the all-frequency specular reflectance further into three separate layers, responsible for high, mid and low frequency reflectance respectively. These separate layers are then computed using a different technique that is appropriate for the type of appearance they are responsible for. Figure 8 shows renderings of the Master Chief in each layer separately and then with all layers combined. The details of the layers are described as follows below.

### 1.4.1 Analytical Specular

This layer is rendered by evaluating the BRDF model directly in shader from the directional light coming from the dominant light direction (See Appendix A for a HLSL

listing). This layer is responsible for capturing the sharp specular highlights as seen in Figure 8 (a). The specular lobe of the Cook Torrance model is the following:

$$R_m(V,L) = \frac{DG}{\pi(N \cdot L)(N \cdot V)}\ , \qquad (6)$$

where *D* is the micro-facet distribution function, and *G* is the geometry attenuation factor, and *L* is the light direction. Interested readers could find how the micro-facet distribution function and the geometry attenuation factor are defined in [COOKTORRANCE81]. For this layer, we only consider the light from the dominant light direction, which can be approximated by fitting a directional light to the quadratic SH light vector. For the direction vector, we use the optimal linear direction (the SH linear coefficients normalized). This direction is well behaved and smoothly varying. Alternatively, we can store a separate texture that encodes the dominant light direction. However, care must be taken to ensure that the directions are filterable since these textures are subject to various filtering operations in the texturing hardware. We can find the dominant light intensity by minimizing the squared error $E$ between the SH light and the SH version of the "directional" light as follows:

$$E = \sum_{i=0,\ldots 8} (\lambda_i - cY_i(d))^2 .$$

Let the derivative with respect to d be zero, $E' = 0$ , then the intensity of the directional light is given by

$$c = \sum_{i=0,\ldots 8} (\lambda_i Y_i(d)) / \sum_{i=0,\ldots 8} Y_i(d)^2$$

.

We can pre-compute and store the intensity $c$ as a separate HDR texture.

### 1.4.2 Environment Map

For the middle frequency glossy reflectance, we use environment cube-maps that are pre-generated at discrete locations throughout the scene. A rendering of the Master Chief with environment maps only can be seen in Figure 8 (b). Since the analytical specular already handles the sharpest highlights, we render the environment maps with the high frequency filtered out. This also allows us to use smaller cube-map sizes (128 × 128 × 6). Unlike analytical specular, the environment map can capture reflectance from the whole lighting environment.

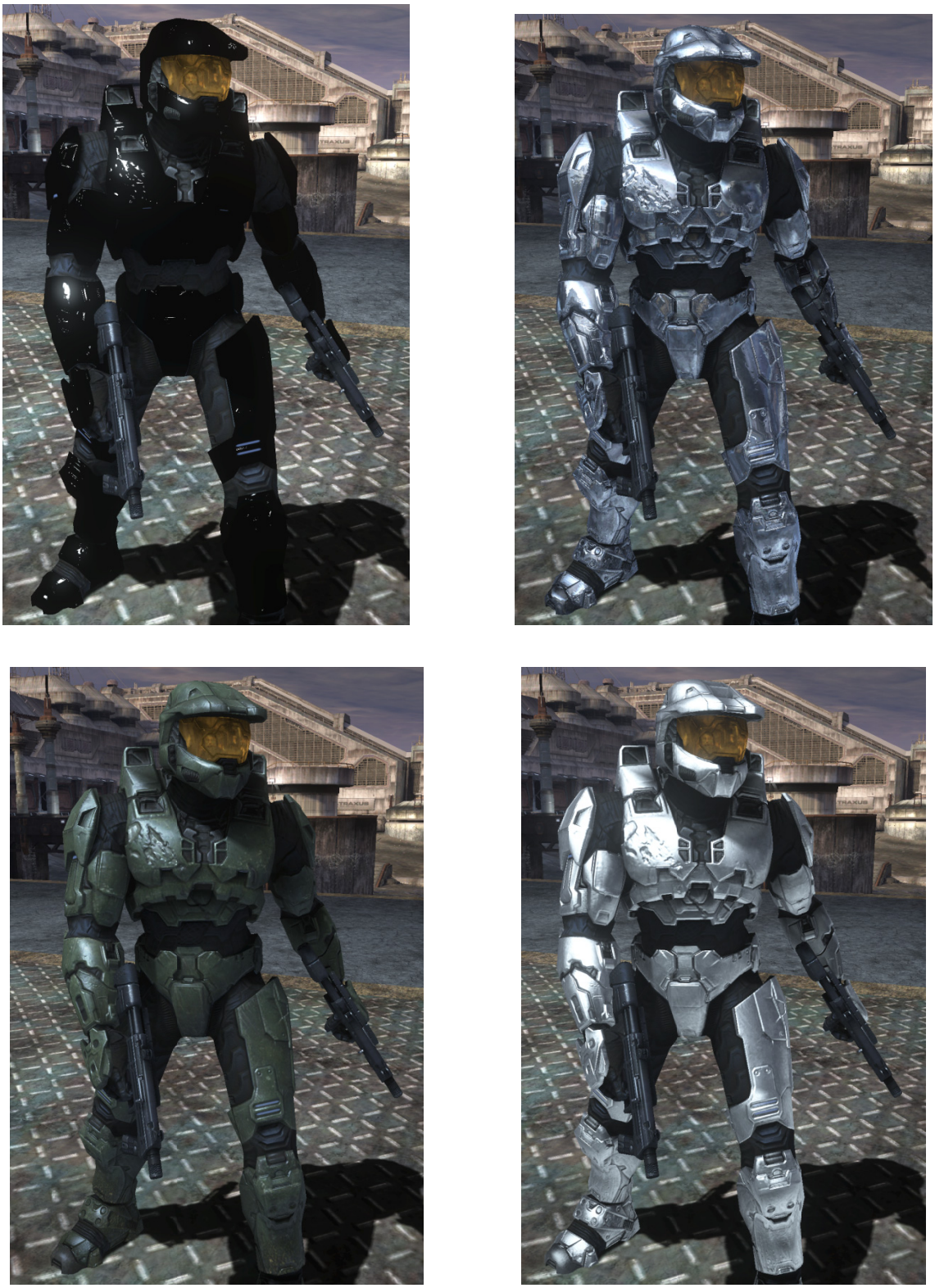

***Figure 8.*** *Rendering of the* Master Chief *showing separate specular layers. Clockwise from top left: (a) Analytical specular only. (b) Environment map only. (c) Area specular only. (d) All combined + diffuse.*

Since we are storing the cube-maps at discrete locations instead of per-pixel on the surface, the environment map sampling suffers from the same limitation mentioned earlier, i.e. the inability to handle sharp transitions along the shadow boundaries. To overcome this, we pre-divide the environment maps by the area specular (described below), and in the shader, we multiply back in the area specular. This simple scheme allows us to darken or brighten the environment maps in and out of shadows.

### 1.4.3 Area Specular

We call the low frequency layer of the glossy reflectance the area specular term. This can be seen in Figure 8 (c). We developed a novel method to parameterize the Cook Torrance BRDF model in SH, which can then be rendered directly from the SH light map representation or any arbitrary lighting that can be represented in the SH basis.

Our method differs from others in that we parameterize the whole BRDF model instead of each material. Therefore we can freely change any parameters of the BRDF model in real time and generate different materials. Spatially varying BRDF can be achieved by storing these parameters in a texture. Our model requires only 3K of storage and can be computed efficiently in real time. Our method can be extended to other BRDF models, and the Phong BRDF is shown in the Appendix as an example.

#### 1.4.3.1 *Light Integration*

Recall equation 3, the specular component of the Cook Torrance BRDF is:

$$I_s(V) = k_s \oint FR_m(V,L)\cos(\theta)\ell(\omega)d\omega$$

If we project both the BRDF and the cosine term in SH, then by the convolution theorem the integral becomes a SH dot product. We define the SH projection of the BRDF as $B_{m,i}(V)$. We also divide the Fresnel term by $F_0$ whose purpose will be explained later.

$$B_{m,i}(V) = \oint \frac{F}{F_0} R_m(V,L)\cos(\theta)Y_i(\omega)d\omega \quad (7)$$

$$I_s(V) = k_s F_0 \sum_{i=0}^{8} \lambda_i B_{m,i}(V) \quad (8)$$

#### 1.4.3.2 *Fresnel Approximation*

We first introduce two terms for convenience:

$$C_{m,i}(V) = \oint R_m(V,L)\cos(\theta)Y_i(\omega)d\omega$$
$$D_{m,i}(V) = \oint \left(1-(L\cdot H)^5\right)R_m(V,L)\cos(\theta)Y_i(\omega)d\omega$$

The first term denotes the integral of the specular reflection function with the SH basis functions, while the second term is defined based on the Fresnel approximation method by Schlick [SCHLICK94].

For constant Fresnel, $F \approx F_0$, then $B_{m,i}(V) = C_{m,i}(V)$, and the Equation 8 becomes:

$$I_s(V) = k_s F_0 \sum_{i=0}^{8} \lambda_i C_{m,i}(V)$$

For non-constant Fresnel, we can use the Schlick approximation [SCHLICK94]:

$$F \approx F_0 + (1-F_0)\left(1-(L\cdot H)^5\right).$$

Then

$$B_{m,i}(V) = C_{m,i}(V) + \tfrac{1-F_0}{F_0} D_{m,i}(V),$$

and

$$I_s(V) = k_s F_0 \sum_{i=0}^{8} \lambda_i \left(C_{m,i}(V) + \tfrac{1-F_0}{F_0} D_{m,i}(V)\right)$$

In the following we show how $C_{m,i}(V)$ and $D_{m,i}(V)$ can be pre-integrated offline.

### 1.4.3.3 *Pre-Integration*

By the isotropic property of the Cook Torrance reflectance model, we can always integrate the shading result of Equation 3 in a local frame such that the view direction *V* lies in the *X-Z* plane of the local frame. Therefore we only need to pre-integrate $C_{m,i}(V)$ and $D_{m,i}(V)$ for some view directions in the *X-Z* plane.

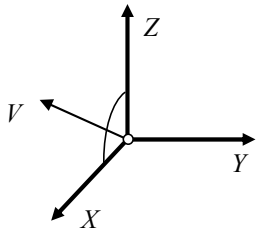


***Figure 9.*** *The local coordinate frame where Z points along the normal, and V lies in the X-Z plane.*

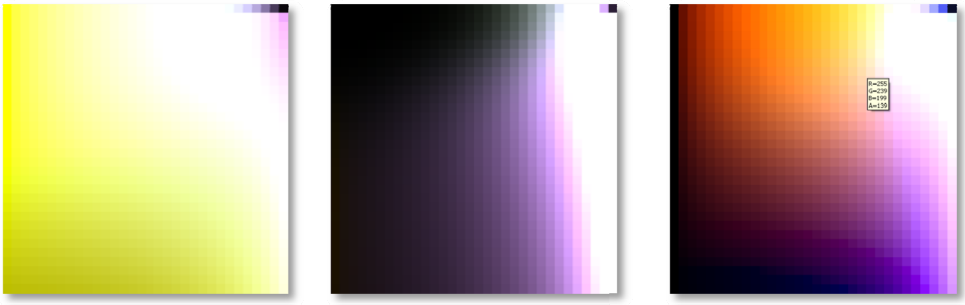

***Figure 10.*** *Pre-integrated C and D textures, from left to right, C(0,2,3,6), D(0,2,3,6)and CD(7,8). Horizontal axis represents viewing changes and the vertical axis roughness changes.*

By the reflective symmetry property of the Cook Torrance reflectance model, we know that $C_{m,i}(V)$ and $D_{m,i}(V)$ are all zero for $i = 1,4,5$, and the variation in the textures are very smooth in the direction of both *m* and *V*. Therefore, we pre-integrate $C_{m,i}(V)$ and $D_{m,i}(V)$ for 16 roughness ($m$) values in (0, 1) and 8 viewing directions in the *X-Z* plane, yielding 12 2*D* textures. The texture values are all in (-2, 2), so we quantize them by 16 bits, or store them in 16-bits *float* format. The total storage is about 16*8*6*2*2 = 3 KB.

Figure 3 shows the integrated C and D textures. With these lookup tables, we can now render any Cook Torrance BRDF in real time and the only storage required is the parameters themselves.

To render area specular, we first construct a local frame as in Figure 9. The lighting SH vector is then rotated into this local frame. To compute the glossy reflectance, we look up the pre-integrated C and D textures, which we then integrate with the lighting through an SH dot product. The HLSL code is listed in the appendix.

## 1.5 Conclusion

In this chapter we have described the core lighting and material models used in Halo 3. The SH light map representation is a natural extension to an existing light mapping pipeline. By storing incident radiance as a SH vector, instead of exit radiance as a color, we can capture sharp shadows just like light maps, and bump maps can be rendered directly from it. SH light maps require much larger storage than traditional light maps, however, lighting data can be heavily compressed with minimal loss of perceptible quality ([HuWang08] describes our multi-stage compression algorithm for SH light maps).

By separating the all frequency reflectance into layers, each of which can then be rendered using different real time techniques, we can preserve the realism of a real word material while keeping the computational and storage costs to manageable levels for a real time game.

We introduced a novel way of parameterizing the Cook Torrance BRDF model in spherical harmonic basis, which can then be rendered under global and environment lighting or any lighting that can be represented in SH. Using small 2D textures as look up tables, all parameters of the BRDF models are free parameters, and we can render any Cook Torrance material efficiently and without per material storage other than the parameters themselves. Spatially varying materials are also possible by putting the parameters in a texture.

Finally, we show that advanced lighting and material models are feasible in the current generation of real time hardware, and with the projected increase of computational power of future generations, lighting and material improvements in games will continue and will result in significant improvements in the realism of games.

## 1.6 Acknowledgements

The Halo 3 graphics engine was developed by a team of dedicated engineers and researchers, they are:

Bungie Graphics Team: Hao Chen, Ben Wallace, Chris Tchou, David Cook, Xi Wang. And Microsoft Research Asia: Xinguo Liu, Yaohua Hu, Zhipeng Hu, Kun Zhou, Minmin Gong.

The authors would like to thank the following people for their contribution to the Halo 3 graphics engine: Peter Pike Sloan, Baining Guo, Harry Shum, Kutta Srinivasan, Matt Lee, Mikey Wetzel.

## 1.7 References:

[BASRIJACOBS03] BASRI, R., AND JACOBS, D. W. 2003. Lambertian reflectance and linear subspaces. *IEEE Trans. Pattern Anal. Mach. Intell. 25*, 2, pp. 218–233.

[BLINN77] BLINN, J. F. 1977. Models of light reflection for computer synthesized pictures. *ACM SIGGRAPH Comput. Graph. 11*, 2, pp. 192–198.

[CHEN08] CHEN, H. Lighting and materials of *Halo 3*. Game Developers Conference, 2008.

[COOKTORRANCE81] COOK, R. L., AND TORRANCE, K. E. 1981. A reflectance model for computer graphics. In *Proceedings of ACM SIGGRAPH 1981*, pp. 307–316.

[GOODTAYLOR05] GOOD, O., AND TAYLOR, Z. 2005. Optimized photon tracing using spherical harmonic light maps. In *Proceedings of ACM SIGGRAPH 2005, Technical Sketches*, p. 53.

[GSHG98] GREGER, G., SHIRLEY, P., HUBBARD, P. M., AND GREENBERG, D. P. 1998. The irradiance volume. *IEEE Comput. Graph. Appl. 18*, 2, pp. 32–43.

[HUWANG08] HU, Y., AND WANG, X. Lightmap compression in Halo 3. Game Developers Conference, 2008.

[ICG86] IMMEL, D. S., COHEN, M. F., AND GREENBERG, D. P. 1986. A radiosity method for non-diffuse environments. *ACM SIGGRAPH Comput. Graph. 20*, 4, pp. 133–142.

[KAJIYA86] KAJIYA, J. T. 1986. The rendering equation. In *Proceedings of ACM SIGGRAPH 1986*, pp. 143–150.

[KSS02] KAUTZ, J., SLOAN, P.-P., AND SNYDER, J. 2002. Fast, arbitrary brdf shading for low-frequency lighting using spherical harmonics. In *Proceedings of the 13th Eurographics workshop on Rendering 2002*, pp. 291–296.

[NDM05] NGAN, A., DURAND, F., AND MATUSIK, W. 2005. Experimental analysis of brdf models. In *Proceedings of the Eurographics Symposium on Rendering 2005*, pp. 117–226.

[OAT05] OAT, C. Irradiance Volumes for Games, Game Developers Conference, 2005. http://ati.amd.com/developer/gdc/GDC2005_PracticalPRT.pdf

[PSS99] PREETHAM, A.J., SHIRLEY, P. AND SMITS, B. 1999. A Practical Analytic Model for Daylight, In Proceedings of Siggraph 1999, pp. 91 – 100, Los Angeles, CA.

[RAMAMOORTHIHANRAHAN01] RAMAMOORTHI, R., AND HANRAHAN, P. 2001. An efficient representation for irradiance environment maps. In *Proceedings of ACM SIGGRAPH 2001*, pp. 497–500.

[RAMAMOORTHIHANRAHAN01B] RAMAMOORTHI, R., AND HANRAHAN, P. 2001. On the relationship between radiance and irradiance: Determining the illumination from images of a convex Lambertian object. *Journal of the Optical Society of America, Vol. 18*, 10, pp. 2448–2459.

[RAMAMOORTHIHANRAHAN02] RAMAMOORTHI, R., AND HANRAHAN, P. 2002. Frequency space environment map rendering. In *Proceedings of ACM SIGGRAPH 2002*, 517–526.

[SCHLICK94] SCHLICK, C. 1994. An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forums. 13*, (3), 233–246.

[SLOANSNYDER02] SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low frequency lighting environments. *ACM Trans. Graph. 21*, 3, 527–536.

[SHHS03] SLOAN, P.-P., HALL, J., HART, J., AND SNYDER, J. 2003. Clustered principal components for precomputed radiance transfer. *ACM Trans. Graph. 22*, 3, 382–391.

[VILLEGASSEAN08] VILLEGAS, L., AND SEAN S. Life on the Bungie Farm: Fun Things to Do with 180 Servers . Game Developers Conference, 2008.

# Appendix A. Shader Code Listings:

```
float3 diffuse_reflectance(float3 normal, float4 lighting_constants[10])
{
  float c1 = 0.429043f;
  float c2 = 0.511664f;
  float c4 = 0.886227f;
  float3 x1, x2, x3;

  //linear
  x1.r = dot( normal, lighting constants[1].rgb);
  x1.g = dot( normal, lighting constants[2].rgb);
  x1.b = dot( normal, lighting constants[3].rgb);

  //quadratic
  float3 a = normal.xyz*normal.yzx;
  x2.r = dot( a.xyz, lighting constants[4].rgb);
  x2.g = dot( a.xyz, lighting constants[5].rgb);
  x2.b = dot( a.xyz, lighting_constants[6].rgb);

  float4 b = float4( normal.xyz*normal.xyz, 1.f/3.f );
  x3.r = dot( b.xyzw, lighting constants[7].rgba);
  x3.g = dot( b.xyzw, lighting constants[8].rgba);
  x3.b = dot( b.xyzw, lighting constants[9].rgba);

  float3 lightprobe_color=
     c4 * lighting_constants[0] + (-2.f*c2) * x1 + (-2.f*c1)*x2 - c1 * x3;O)

  return lightprobe color/3.1415926535f;
}

void pack_constants(in float3 sh[9], out float4 lc[10])
{
  lc[0]= float4(sh[0], 0);
  lc[1]= float4(sh[3].r, sh[1].r, -sh[2].r, 0);
  lc[2]= float4(sh[3].g, sh[1].g, -sh[2].g, 0);
  lc[3]= float4(sh[3].b, sh[1].b, -sh[2].b, 0);
  lc[4]= float4(-sh[4].r,sh[5].r, sh[7].r, 0);
  lc[5]= float4(-sh[4].g,sh[5].g, sh[7].g, 0);
  lc[6]= float4(-sh[4].b,sh[5].b, sh[7].b, 0);
  lc[7]= float4(-sh[8].r, sh[8].r,-sh[6].r*1.7320508f,
                sh[6].r*1.7320508f);
  lc[8]= float4(-sh[8].g, sh[8].g,-sh[6].g*1.7320508f,
                sh[6].g*1.7320508f);
  lc[9]= float4(-sh[8].b, sh[8].b,-sh[6].b*1.7320508f,
                sh[6].b*1.7320508f);
}
```

***Listing 1.*** *HLSL shader code for rendering diffuse reflectance from SH Light map. Notice a quadratic SH vector is pre-packed into 10* `float4` *constants.*

```
void calc_material_analytic_specular_cook_torrance_ps(
  in float3 view dir,
  in float3 normal dir,
  in float3 reflect dir,
  in float3 light dir,
  in float3 light_intensity,
  in float3 c_fresnel_f0,
  in float  c toughness,
  out float3 analytic specular)
{
  float n dot l = dot( normal dir, light dir );
  float n_dot_v = dot( normal_dir, view_dir );
  float min_dot = min( n_dot_l, n_dot_v );
  if ( min dot > 0)
  {
    // geometric attenuation
    float3 half vector = normalize( view dir + light dir );
    float n_dot_h = dot( normal_dir, half_vector );
    float v_dot_h = dot( view_dir, half_vector);
    float G = 2 * n dot h * min dot / (saturate(v dot h));

    //calculate fresnel term
    float3 f0= c fresnel f0;
    float3 sqrt_f0 = sqrt( f0 );
    float3 n = ( 1.f + sqrt_f0 )/( 1.0 - sqrt_f0 );
    float3 g = sqrt( n*n + v dot h*v dot h - 1.f );
    float3 gpc = g + v dot h;
    float3 gmc = g - v dot h;
    float3 r =(v dot h*gpc-1.f)/(v dot h*gmc+1.f);
    float3 F= (0.5f*( (gmc*gmc)/(gpc*gpc+0.00001f))*( 1.f+r*r));

    //calculate the distribution term
    float t roughness= c toughness;
    float m squared= t roughness*t roughness;
    float cosine alpha squared = n dot h * n dot h;
    float D;
    D= exp((cosine_alpha_squared-1)/
           (m squared*cosine alpha squared))/
           (m squared*cosine alpha squared*cosine alpha squared);

    //puting it all together
    analytic_specular= D*saturate(G)/(3.14159265 * n_dot_v)*F;
    analytic_specular*= light_intensity;
  }
  else
  {
    analytic specular= 0.0f;
  }
}
```

***Listing 2.*** *Cook Torrance BRDF evaluated directly in shader from a point light source for the analytical specular. The dominant light is a directional light fitted from the quadratic SH coefficients.*

```
void area_specular_cook_torrance(
  in float3 view_dir,
  in float3 rotate z,
  in float4 sh 0,
  in float4 sh 312[3],
  in float4 sh 457[3],
  in float4 sh_8866[3],
  in float roughness,
  in float r dot l,
  out float3 area specular)
{
  float3 specular part;
  float3 schlick_part;
  //build the local frame
  float3 rotate x= normalize(view dir-dot(view dir,rotate z)*rotate z);
  float3 rotate y= cross(rotate z, rotate x);

  //calculate the texure coord for lookup
  float2 view_lookup= float2(dot(view_dir,rotate_x), roughness);

  // bases: 0,2,3,6
  float4 c value= tex2D(g sampler cc0236, view lookup);
  float4 d value= tex2D(g sampler dd0236, view lookup);

  //rotate lighting basis 0,2,3,6 into local frame
  float4 quadratic_a, quadratic_b, sh_local;
  quadratic a.xyz= rotate z.yzx * rotate z.xyz *(-SQRT3);
  quadratic b= float4(rotate z.xyz*rotate z.xyz,1.0f/3.0f)*0.5f*(-SQRT3);

  //red
  sh_local.xyz= sh_rotate_023( 0,
                               rotate_x,
                               rotate z,
                               sh 0,
                               sh 312);
  sh local.w= dot(quadratic a.xyz,
         sh_457[0].xyz)+dot(quadratic_b.xyzw, sh_8866[0].xyzw);

  //dot with C and D look up
  sh local*= float4(1.0f, r dot l, r dot l, r dot l);
  specular part.r= dot(c value, sh local );
  schlick_part.r= dot(d_value, sh_local );

  //repeat for green and blue
      …
```

***Listing 3 Part 1.*** *Shader code for area specular calculation, continues next page*

```
  // basis - 7
  c value= tex2D( g sampler c78d78, view lookup ).SWIZZLE;
  quadratic a.xyz = rotate x.xyz * rotate z.yzx + rotate x.yzx *
                    rotate z.xyz;
  quadratic b.xyz = rotate x.xyz * rotate z.xyz;
  sh_local.rgb= float3(dot(quadratic_a.xyz, sh_457[0].xyz) +
                       dot(quadratic_b.xyz, sh_8866[0].xyz),
                       dot(quadratic a.xyz,sh 457[1].xyz) +
                       dot(quadratic b.xyz, sh 8866[1].xyz),
                       dot(quadratic a.xyz, sh 457[2].xyz) +
                       dot(quadratic b.xyz, sh 8866[2].xyz));


  sh local*= r dot l;
  //c7 * L7
  specular part.rgb+= c value.x*sh local.rgb;
  //d7 * L7
  schlick_part.rgb+= c_value.z*sh_local.rgb;

  //basis - 8
  quadratic a.xyz = rotate x.xyz * rotate x.yzx - rotate y.yzx *
                    rotate y.xyz;
  quadratic b.xyz = 0.5f*(rotate x.xyz * rotate x.xyz - rotate y.xyz *
                    rotate_y.xyz);
  sh_local.rgb= float3(-dot(quadratic_a.xyz, sh_457[0].xyz) -
                        dot(quadratic b.xyz, sh 8866[0].xyz),
                       -dot(quadratic a.xyz, sh 457[1].xyz) -
                        dot(quadratic b.xyz, sh 8866[1].xyz),
                       -dot(quadratic a.xyz, sh 457[2].xyz) -
                        dot(quadratic_b.xyz, sh_8866[2].xyz));
  sh_local*= r_dot_l;

  //c8 * L8
  specular part.rgb+= c value.y*sh local.rgb;
  //d8 * L8
  schlick_part.rgb+= c_value.w*sh_local.rgb;
  schlick_part= schlick_part * 0.01f;

  area specular= specular part*k f0 + (1 - k f0)*schlick part;

}

float3 sh_rotate_023(int irgb, float3 rotate_x, float3 rotate_z,
                     float4 sh 0,
                     float4 sh 312[3])
{
  float3 result = float3( sh 0[irgb],
                         -dot(rotate_z.xyz, sh_312[irgb].xyz),
                          dot(rotate_x.xyz, sh_312[irgb].xyz));

  return result;
}
```

***Listing 3 Part 2.*** *Shader code for area specular calculation*

## Appendix B. Phong BRDF

Let $\hat{V}$ be the reflection direction of the viewing direction around the surface normal. We define a glossy reflection distribution function as follows:

$$R_m(V,L) = \frac{1}{\cos\theta_V}\frac{1}{\pi m^2 \cos^3\alpha}\exp\left\{-\frac{\tan^2\alpha}{m^2}\right\},$$

where $\alpha$ is the angle between the reflection direction $\hat{V}$ and the incident light direction $L$. Assume that the effective incoming light has almost the same incident angle as $\theta_V$, the total energy is conserved, since

$$\oint \frac{1}{\pi m^2 \cos^3 \alpha} \exp\left\{-\frac{\tan^2 \alpha}{m^2}\right\} d\omega = 1.$$

Under an environmental lighting $L(\omega)$, the illumination result can be computed by:

$$I(V) = \oint \frac{1}{\cos \theta_V} \frac{1}{\pi m^2 \cos^3 \alpha} \exp\left\{-\frac{\tan^2 \alpha}{m^2}\right\} \ell(\omega) \cos(\theta) d\omega$$

Again assume that the effective incoming light has almost the same incident angle as $\theta_V$ (i.e. the material is high glossy), we have

$$I(V) = \oint \frac{1}{\cos \theta_V} \frac{1}{\pi m^2 \cos^3 \alpha} \exp\left\{-\frac{\tan^2 \alpha}{m^2}\right\} \ell(\omega) \cos(\theta_V) d\omega$$

$$= \oint \frac{1}{\pi m^2 \cos^3 \alpha} \exp\left\{-\frac{\tan^2 \alpha}{m^2}\right\} \ell(\omega) d\omega = \sum_i \lambda_i E_{m,i}(V)$$

where

$$E_{m,i}(V) = \oint \frac{1}{\pi m^2 \cos^3 \alpha} \exp\left\{-\frac{\tan^2 \alpha}{m^2}\right\} Y_i(\omega) d\omega$$

(note that angle $\alpha$ has dependence on $V$.)

# Appendix C. Additional Screen Captures.

# Chapter 2

# Advanced Virtual Texture Topics

Martin Mittring[1]
Crytek GmbH

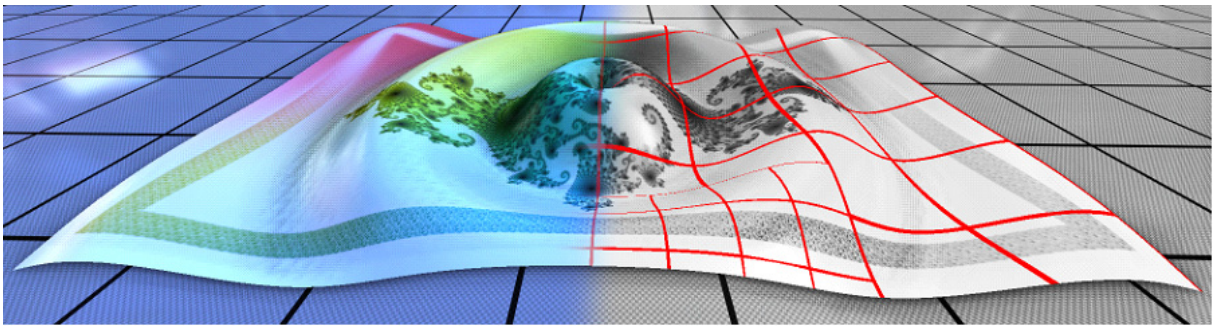

***Figure 1.*** *A visual representation of a virtual texture*

## 2.1 Abstract

A *virtual texture*[2] is a mip-mapped texture used as cache to allow a much higher resolution texture to be emulated for real-time rendering, while only partly residing in texture memory. This functionality is already accessible with the efficient pixel shader capabilities available on the recent generations of commodity GPUs. In this chapter we will be discussing technical implications on engine design due to virtual textures use, content creation issues, results, performance and image quality. We will also cover several practical examples to highlight the challenges and to offer solutions. These include texture filtering, block compression, float precision, disk streaming, UV borders, mip-map generation, LOD selection and more.

[1] Martin@Crytek.de
[2] The term is derived from the OS/CPU feature "virtual memory", which allows transparent memory access to a larger address space than the physical memory.

## 2.2 Motivation

The diagram in *Figure* shows how different hardware devices for texture storage can be classified with a different speed/amount ratio. Caching is a common technique to allow fast access to larger data set to live in slower memory. The virtual texture described here uses using traditional texture mapping to cache data coming from the respective slower content device.


***Figure 2.*** *Hardware can be classified depending on a speed/amount ratio.*

Note: On the GPU a texture lookup operation is limited to one texture only and random access to the whole video memory is not possible, limited in size[3] or high latency. Because of that the diagram in *Figure* lists "Texture" and "Video memory" as separate units.

### 2.2.1 Texture Streaming is Becoming a Necessity

Texture mapping is common-place and highly efficient on consumer GPUs for over a decade. Many challenges have been solved by hardware support for mip-mapping, advanced texture filtering, border clamp/mirror rules and compressed texture formats. Modern real-time rendering engines are faced with another challenge: Screen resolution and higher quality standards now require high resolution textures and for draw call efficiency it's even advised to share one texture for multiple objects [NVIDIA04]. Some graphics hardware already supports texture resolutions up to 8K (8192), but that might not be enough for some applications, and, more of a problem, the memory requirements grow rapidly with texture size. Because the simulated world size is also expected to be much larger it's no longer possible to keep all textures in graphic card memory (a typical limit is 512MB) and not even in main memory. Having more main memory doesn't help when limited being by the 32 bit address space (2GB on typical 32bit OS). A 64 bit OS allows using more main memory but most installed OS and

[3] The maximum size of a texture can be a limiting factor (usually from 1024 to 8192 depending on a specific hardware generation).

hardware is still 32 bit. Limited amount of physical memory can be compensated by using virtual memory from hard drive. Unfortunately this option is not viable for real-time rendering as traditional virtual memory (as an OS and hardware feature) stalls until the request is resolved. The situation is exacerbated without an available hard-drive as it might be the case on consoles. To overcome this and to get a fast level loading time modern engines are required to do texture streaming.

### 2.2.2 Problems with Per Mip-Map Texture Streaming

We can avoid stalling while requesting to load a specific texture mip level from the hard-drive by using a lower mip level as a fallback until the desired level is uploaded to the graphics card. This is acceptable for real-time games and the lower resolution texture can go unnoticed with sufficient care. Unfortunately it's basically impossible to add or remove a mip-map dynamically. The Direct3D9 function `SetLOD()` was made for that but that only affects the video memory alone and doesn't change the issue of the physical and virtual memory. Most hardware keeps all mip-maps in one block of continuous memory and updating a single mip-level becomes a full mip chain update. In Crysis™ (**Error! Reference source not found.**) we wanted to save virtual memory so to adjust the mip-level we had to create and release textures at runtime. That is very bad for stable performance and MultiGPU (SLI/Crossfire) scaling but it was a manageable solution at the time. Streaming allowed us to stay within the 32 bit limits with run-time data requirement sometimes exceeding the limits. On 64 bit and enough main memory or when using half resolution textures the texture streaming is not necessary and performance is more stable.

***Figure 3**. The screenshot from the game* Crysis™ *shows the need for texture streaming: large rich environments with many details.*

Avoiding `Create()` and `Release()` calls at runtime is possible when textures can be reused, but only if texture formats and sizes match. This is very restricting and even wasteful on the memory usage, and therefore not a practical option. As a result, this problem is a serious issue for game developers and deserves better API and hardware support.

The idea of virtual textures is to manage the texture memory at a different granularity than the mip-map level. Often only small areas are required in each mip-map and thus uploading a full mip-map would be wasteful. It's much more efficient if we only upload the actually required portions. The virtual texture method itself is simple, but it has a lot of interesting related topics attached to it which we will discuss here after explaining the basic method itself.

## 2.3 Implementing Virtual Textures with Pixel Shaders

### 2.3.1 Virtual Texture – A Definition

For the virtual texture we only keep relevant parts of the texture in fast memory and asynchronously request missing parts from a slower memory (while using the content of a lower mip-map as fall-back). This implies that we need to keep parts of lower mip-maps in memory and these parts need to be loaded first. To allow efficient texture lookups coherent memory access is achieved by slicing up the mip-mapped texture into reasonably sized pieces. Using a fixed size for all pieces (now called texture tiles) the cache can be managed more easily and all tile operations, e.g. reading or copying, have constant time and memory characteristics. Mip-maps smaller than the tile size are not handled by our implementation. This is often acceptable for applications like terrain rendering where the texture is never that far away and little aliasing is acceptable. If required, this can be solved as well by simply packing multiple mip-maps into one tile. For distant objects it's even possible to fall back to normal mip-mapped texture mapping, but that requires a separate system for managing that.

As shown in Figure 4, a typical virtual texture is created from some source image format at preprocessing time without imposing the API and graphics hardware limits on texture size. The data is stored on any lower access speed device, such as hard drive. Unused areas of the virtual texture can be dropped (saving memory as a nice side bonus). The texture in the video memory (now called *tile cache*) consists of tiles required to render the 3D view. We also need the indirection information in order to efficiently reconstruct the virtual texture layout. Both the tile texture cache and the indirection texture are dynamic and adapt to one or multiple views.

To manage our virtual texture we use a quad-tree because all required operations can be implemented in constant time. Here the state of the tree represents the currently used texture tiles for a virtual texture. All nodes and leaves are associated with a texture

tile and in a basic implementation only the highest available resolution in the quad-tree is used. The lower resolution data is stored as fall-back when we drop some leafs and can also be used to fade in higher resolution texture tiles gradually. We refine or coarsen the virtual texture only at the leaf level. In addition to the quad-tree we need additional code implementing the cache strategy (e.g. Least Recently Used).

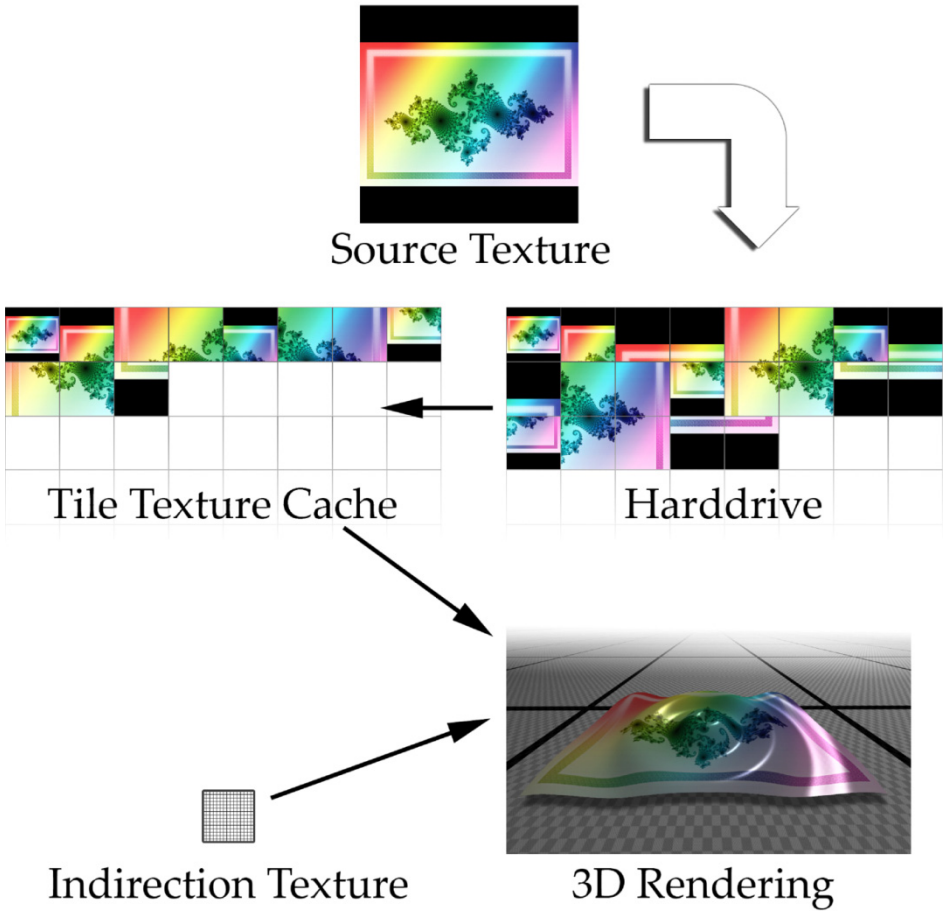


***Figure 4.*** *Typical usage scenario of the virtual texture method*

### 2.3.2 Reconstruction in the Pixel Shader

The reconstruction needs to be very efficient. While some applications allow efficient indirections per draw call[4] or pre-split geometry[5], this is too limiting in general and hard to implement efficiently for general geometry. Simply using a pixel shader to implement this functionality is straight-forward and intuitive. This code returns the texture

[4] Games like Far Cry™ or Crysis™ render one terrain sector with texture tile per draw call. That allows the tile cache to be split in individual texture and that simplifies tile updates.
[5] It's possible to setup the vertex texture coordinates to render multiple tiles in one draw call.

coordinates in the tile cache texture for a given virtual texture coordinate. This even allows emulating texture coordinate addressing modes like “warp” or “mirror”.

Here we assume at least 32 bit float precision in the computations (which is not supported by older pixel shader versions, but is common by latest generations of DirectX® 9.0c-capable graphics hardware). Note that on DirectX® 9 you have to offset your texture lookups by half a texel. In OpenGL you have to do similar computations.

Efficiency is very important as this code is executed for every pixel. This is why the quad-tree traversal in the pixel shader is replaced by a single unfiltered texture lookup. This texture (now called indirection texture) can be quite small in memory and because of the coherent texture lookups it is also very bandwidth friendly. A single texture lookup allows computing the texture coordinates in the tile cache with simple math in constant time.

The idea of implementing a virtual texture using a pixel shader received a lot of attention after John Carmack mentioned the “Mega texture” technique he has been working on (as described in [IDTECH507]). The technique is used in the commercial product “Quake Wars” and is currently developed to a more generalized solution at `id` Software. The basic idea is clear, but implementation details are not described. Sean Barrett investigated further and shared his knowledge and his version of the method at GDC 2008. It’s an advised read for anyone that wants to implement it [BARRETT08].

The following HLSL code can be used in the pixel shader to compute for the given virtual texture coordinate the associated tile cache texture coordinate:

```
float4 g_vIndir;             // w,h,1/w,1/h indirection texture extend
float4 g_Cache;              // w,h,1/w,1/h tile cache texture extend
float4 g_CacheMulTilesize;   // w,h,1/w,1/h tile cache texture extend
                             //             * tilesize

sampler IndirMap = sampler_state
{
  Texture   = <IndirTexture>;
  MipFilter = POINT;
  MinFilter = POINT;
  MagFilter = POINT;
//  MIPMAPLODBIAS = 7;       // using mip-mapped indirection texture,
                             // 7 for 128x128
};

float2 AdjustTexCoordforAT( float2 vTexIn )
{
  float fHalf = 0.5f;        // half texel for DX9, 0 for DX10

  float2 TileIntFrac = vTexIn*g_vIndir.xy;
  float2 TileFrac = frac(TileIntFrac)*g_vIndir.zw;
  float2 TileInt = vTexIn - TileFrac;
  float4 vTiledTextureData = tex2D(IndirMap,TileInt+fHalf*g_vIndir.zw);

  float2 vScale = vTiledTextureData.bb;
  float2 vOffset = vTiledTextureData.rg;

  float2 vWithinTile = frac( TileIntFrac * vScale );

  return vOffset + vWithinTile*g_CacheMulTilesize.zw + fHalf*g_Cache.zw;
}
```

***Listing 1.*** *HLSL Shader Code to compute the texture coordinates in the tile cache texture*

The code can be optimized further but care must be taken to keep the floating point error minimal. The texture cache should not have mip-maps and the lookup should be bilinear only.

In [BARRETT08] Sean Barrett mentions the simplest shader fragment he came up with (with a hint from John Carmack):

```
tex page   , vtc , tex0     , 2D
mad phys.xy, vtc , page.xyxy, page.zwzw
tex color  , phys, tex1     , 2D
```

**Listing 2.** *Pixel shader assembler code to compute the texture coordinates with the tile cache lookup*

This would be 2 instructions only for the texture coordinate computation. We haven't tried that but float precision might be an issue, especially when using 64 bit or even 32 bit textures - using a 128 bit texture may be slower on some hardware.

#### 2.3.2.1 *The Indirection Texture*

The indirection texture can be easily generated from the data in the quad-tree. With a single unfiltered lookup into the indirection texture and simple math with constants we can compute the texture coordinates in the tile cache. The values stored in a texel contain the scale of the tile and the 2D offset in tile cache. In our implementation we use a 64 bit texture with FP16 channels. Using a 32 bit texture format with 8 bit channels is possible but you have to adjust the values in the pixel shader with additional instructions. Beware that scaling might not return the values you would expect. By storing a value from 0 to 255 in the texture you get values from 0.0 to 1.0 in the pixel shader. This result is a guaranteed. Scaling these values by 255.0 you would think would result in integer values. However, this may not be the case. Floating point math can be an issue, but even worse is that on some hardware the precision seems to be lower, rather comparable to 16 bit floats. In [BARRETT08] the problem was solved by rounding[6], but the author admits that this might be not the most efficient approach.

Here we use a 64 bit (4 channels FP16) texture format as it is compact and doesn't suffer from the issues mentioned before. The memory bandwidth requirements for our use are minimal as the method has a very high texture lookup coherency, i.e. texture cache misses are rare. However depending on the hardware the lookup itself might require multiple cycles on 64 bit or 128 bit texture formats. Integer textures, an integer texture lookup[7] and integer math can be a good choice on some hardware as we want constant precision over whole domain and float is likely to cause problems here. This would also allow using higher resolution texture caches (> 8K) easily.

[6] To get the integer value from of a 8 bit channel this shader code was used: floor(channel*255+0.5)
[7] Direct3D 10 offers the HLSL Load() but only unfiltered

The following C/C++ code can be used to compute the indirection texture content:

```
// float to fp16(s1e5m10) conversion (does not handle all cases)
WORD float2fp16( float x )
{
  uint32 dwFloat = *((uint32 *)&x);
  uint32 dwMantissa = dwFloat & 0x7fffff;
  int iExp = (int)((dwFloat>>23) & 0xff) - (int)0x7f;
  uint32 dwSign = dwFloat>>31;

  return (WORD)( (dwSign<<15)
      | (((uint32)(iExp+0xf))<<10)
      | (dwMantissa>>13) );
}

WORD texel[4];              // texel output
RECT recTile;               // in texels in the tilecache texture
int iLod;                   // 0=full domain, 1=2x2, 2=4x4, ...
int iSquareExtend;          // indirection texture size in texels
float fInvTileCache;        // tile.Width / texCacheTexture.Width

texel[0] = float2fp16(recTile.left*fInvTileCache);
texel[1] = float2fp16(recTile.top*fInvTileCache);
texel[2] = float2fp16((1<<iLod)/ iSquareExtend);

texel[3] = 0;               // unused
```

***Listing 3.*** *C/C++ code to compute the content of the FP16 indirection texture*

Using a mip-mapped indirection texture requires a few more texture update operations but it also allows per-pixel LOD which looks much smoother. The per-pixel LOD code computes a lower (or the same) LOD that is available in the texture tile cache. Standard texture mapping with LOD adjustment[8] can be sufficient but anisotropic texture mapping would provide better quality at steep angles. In *Figure* different texture filtering modes are shown.

The virtual texture technique can be extended to more than two dimensions. By using volume textures or multiple slices in a 2D texture the lookup is still quite efficient. However multidimensional content scales quickly regarding memory demand and then it's better to adapt at a finer granularity, i.e. a smaller tile size is needed. In GPGPU applications (such as [LEFOHN03]) data is often processed in volume textures and often barely fits into video memory. Caching can be done as usual but processing might require the full data set and locally varying LOD is not common and thus dynamic methods are often not used there.

[8] See `MIPMAPLODBIAS` in the pixel shader code

#### 2.3.2.2 *Efficient Filtering Through Borders*

A naive implementation of bilinear filtering requires 4 lookups into the indirection texture, 4 lookups in the tile cache, followed by bilinear interpolation in the shader. While this may be somewhat reasonable for a hardware implementation, in the pixel shader implementation this is wasteful with respect to performance. Adding a small border is much more efficient as the much more efficient built-in hardware bilinear filtering can be used. A one-pixel border is enough to get bilinear filtering on uncompressed textures but in order to add support for DXT compressed textures a 4 pixel border is necessary. This is because the DXT block compression is based on 4x4 blocks and to avoid seams you need to add a full block to the border. Furthermore it's better to center the tile to get more stable results for imprecise texture coordinate computations. That also simplifies the implementation of more advanced filtering like bi-cubic filtering or anisotropic filtering. The later one would be an interesting topic for this chapter but because we haven't done any implementation we skip it here.

Unfortunately the additional borders waste memory, destroy the power-of-two texture extents, and break the memory alignment of the tiles. If you have non-power-of-two tiles it's probably better to add some padding to the tile cache to create the texture with power-of–two dimensions[9]. Otherwise you might be faced with undefined memory and performance characteristics from the APIs and graphic card drivers.

Alternatively, instead of adding the border to the tile we can also reduce the tile size by the border to allow the sum of both to be power of two[10]. This is best for the hardware implementation but resulting visual quality can suffer a lot. That loss is due to aliasing in the mip-maps caused by the down sampling of the source texture to slightly less than its half size. A good down-sampling algorithm can limit the aliasing in the lower mip-maps but even the top mip-map is affected by this design decision and that can be visible especially when using regular patterns in the texture.

#### 2.3.2.3 *Maximum Size of the Virtual Texture*

As mentioned, our implementation is based on a single indirection only and assumes all tiles in the tile cache have the same size you can compute the virtual texture resolution:

$$\text{Resolution}_{\text{virtual texture}} = \text{Resolution}_{\text{Indirection texture}} * \text{Resolution}_{\text{texture tile without border}}$$

Examples: 16k = 128 * 128
65k = 256 * 256
256k = 256 * 1024

[9] e.g. for 7 tiles with 128+4 pixel extend (128+4)*7=924, the next power of two extend would be 1024
[10] e.g. for 8 tiles with 124+4 pixel extend (124+4)*8=1024, but the usable tile size is no longer power of two

Using a larger tile size limits the adaptive property of the method and using a larger indirection texture becomes inefficient when updating the texture, especially with CPU updates. Unfortunately there is another limit for virtual texture resolution. With a typical implementation using floating point math to run on older hardware the precision of the float computations becomes a problem when the virtual texture resolution becomes close to 65K. We may see reasonable performance for colour look-ups; however, bilinear filtering will no longer be efficient. We can alleviate the problem by careful ordering of our floating point operations, however, integer maths avoid this all together.

#### 2.3.2.4 *Storing Different Attributes in the Tile Caches*

You can comprise one tile cache of multiple textures to store attributes like diffuse, specular or normal maps as long they share the same tile positions. After computing the position in the tile cache it can be efficiently used for multiple lookups. With a bigger border you can even use differently sized textures for the attributes you want to store.

#### 2.3.2.5 *Splitting the Tile Cache Over Multiple Textures*

Instead of storing different attributes you can use multiple tile caches to get more cache units, but here a new problem appears. Normal hardware rendering only allows to texture from the same textures in one draw call. Performance can be much worse when trying to overcome this limitation: Texture lookups through texture arrays are a bit slower (Direct3D® 10 only) and the alternative of fetching data from several textures and masking the result is even slower. That's why it's good to keep all tiles required for one draw call in the in the same tile cache.

When using multiple tile caches you might end up with one tile cache overused while another one is underused. Moving objects between different caches might be an option but performance will no longer be constant, no matter what strategy you pick. Grouping specific types of objects (e.g. one tile cache for terrain and one for objects) is the simpler solution.

The maximum texture resolution supported on some hardware limits your tile cache size. Here we have two simple solutions: You can tweak the LOD computation and accept blurrier results or you add another cache in the graphic card memory, between the texture and the main memory. Copying between VRAM and the texture is expected to be fast. By computing the local LOD required as small as possible the tile cache can be kept small (see the LOD computation methods described later). Rendering a view is possible from the main tile cache, changing view angle additionally requires the secondary cache and moving the view position requires secondary cache updates. The

later ones can come from a slow media like the DVD and to minimize latency more cache stages can be done on the hard drive and even in main memory.

#### 2.3.2.6 *Tile Cache Texture Updates*

As already mentioned, to avoid bilinear filtering artefacts with DXT block compressed textures we require an additional border of 4 pixels. Due to the lossy DXT compression we need exactly the same block content when compressing blocks of neighbour tiles; otherwise we might reconstruct wrong colour values and thus resulting in visible seams.

There are three basic methods to update a part of a mip-map from CPU. Depending on the specific graphics API[11] and on the graphics card use and the driver version, performance characteristics may be not clearly defined. This is actually is the major problem of the implementation and on some configurations it might even make the method unusable. Further testing is needed to quantify this claim. Experience shows that such driver issues often get addressed after a major game shipped using the technology.

For the update of the tile cache texture we have some requirements:

- Fast in latency and throughput
- Bandwidth efficient (copy only the required part)
- Small memory overhead
- Updates should happen without stalls but correctly synchronized to get the right texture state
- When updating the content by CPU there should be no copy from GPU memory to CPU memory (discard should be used)
- For fast texturing from the tile cache texture it should be in the appropriate memory layout (swizzled[12]) and memory type (video memory). Note that on some hardware compressed textures are stored in linear form (not swizzled).
- Multiple tile updates should have linear or better performance

All methods require some locking of either a full texture or a part of the texture. The driver might do a full update and you probably would only notice on less powerful hardware or with heavy bus usage. We're still investigating further into this area. To describe the three basic methods we use the Direct3D 9 API.

Method 1: Direct CPU update:

The destination texture needs to be in `D3DPOOL_MANGED` and via the `LockRect()` function a section of the texture is updated. This method wastes main memory, and most likely the transfer is deferred till a draw call is using the texture. This method is simple but likely to be less optimal when compared with the next two methods.

[11] OpenGL, Direct3D® 9, Direct3D® 10 or the APIs used on modern consoles

[12] A swizzled memory layout is a cache friendly layout for position coherent texture lookups.

Method 2: With (small) intermediate tile texture:
Here we need one lockable intermediate texture hat can hold one tile (including border). With the `LockRect()` function this texture is updated as a whole. With the `StretchRect()` function the texture is then copied into the destination texture to replace one tile only. This does not work with compressed textures (DXT) as they cannot be used as render target format. For the `StretchRect()` function the source and the destination requires to be in `D3DPOOL_DEFAULT` and that requires the source to be `D3DUSAGE_DYNAMIC` to be lockable. To find out if the driver supports dynamic textures, you are ought to check the caps bits for `D3DCAPS2_DYNAMICTEXTURES` but according to the list in the DirectX SDK even the lowest SM20 cards support this feature.

Method 3: With (large) intermediate tile cache texture:
This method requires a lockable intermediate texture in `D3DPOOL_SYSTEM` with the full texture cache extend. With `LockRect()` the intermediate texture is updated only where required and a following `UpdateTexture()` function call is transferring the data to the destination texture. `UpdateTexture()` requires the destination to be in `D3DPOOL_DEFAULT`.

#### 2.3.2.7 *Indirection Texture Update*

Once the new tile is in the texture cache the indirection texture can be updated. We wish to make this an efficient operation. The indirection texture requires little memory therefore bandwidth is not a issue. However, using several indirection textures and updating them often can become a performance bottleneck. The texture can be updated from the CPU by locking or uploading a new texture. This can cause irregular performance characteristics on current APIs but at the same time has proven to be an acceptable solution. Locking a resource that is in use by the GPU can definitely produce some hitches unless clever renaming is done on the driver side.
If you choose to have a render target texture format for your indirection texture you can also consider GPU updates triggered by CPU. Updates can be done by draw calls and simple quads can be rendered to the texture to update even large regions efficiently.
There shouldn't be too many updates as tile cache updates should be rare and both rely on each other.

In our implementation the indirection texture still has a channel left and storing a tile blend value is possible. With extra shader cost this allows hiding texture tile replacement by slowly blending tiles in or out. A filtered blend value would be even nicer as it allows hiding the seams between the tiles. However this can hide details and experience showed satisfying results without this feature.

### 2.3.3 Mip-Mapping and Virtual Textures

It's best to generate the mip-maps and do the texture compression in an offline process. This way a high quality implementation can be used and the data is optimally prepared for fast access. This is similar to a normal production pipeline but some details differ.

For efficient streaming the mip-maps are organized on disk as tiles of continues blocks and in that form the data needs to be generated. Most mip-map computation implementations assume that the textures can be fully loaded into memory. In that case, developers may waste memory on a large texel format and buffer duplication. When using huge virtual textures you have to assume that the texture cannot be processed in memory, especially if you want your tools to run on 32 bit OS. Fortunately it's not too difficult to make the mip-map generation code running without keeping the full data in memory.

#### 2.3.3.1 *Out of Core Mip-Map Generation*

Typically the input data is stored on the disk in a standard image format. For efficient processing we convert it into a form that allows fast access for algorithms with read and write operations with strong locality. Here again we can make use of some tile based data layout. The tile size here is not dependent on the virtual texture tile size and the border pixels are undesired because that would add redundancy. Redundant data like this can speed up the processing under certain circumstances, but it might cause other problems in the system.

We need a class that can read a section of the image but completely hiding the tile based data layout. The input area can be defined by any rectangle, even outside of the source domain. Pixels outside of the domain can return the wrapped content or a border colour. The class caches tiles that have been requested recently and it also should supports writing tiles temporary on the hard drive. This is used to store the mip-map levels during processing. Using the same functions to access the source data and the intermediate data simplifies the code a lot.

Generating mip-maps is simple: Generate the lowest mip-map of the image recursively by requesting images one mip-map higher and run your favourite mip-mapping filter (add border depending on kernel). In a second pass the generated tiles are extracted, compressed and stored into the streaming friendly format (add border for bilinear filtering and compression).

#### 2.3.3.2 *Kernel Size for the Mip-Map Generation*

For the mip-map generation you can pick either an even (e.g. 2×2 or 4×4) or an odd sized (e.g. 3×3 or 5×5) kernel to down sample the texture to a quarter (half in both

dimensions). This affects the result a lot and that it even has implications on the virtual texture pixel shader implementation.

The **even-sized kernel** is used for normal hardware mip-mapped texture mapping and is therefore good to be used if a consistent look is important. This might be the case if you want to render some object, maybe depending on distance, without the virtual textures.

Example: Box 2×2:

| | |
|---|---|
| 1/4 | 1/4 |
| 1/4 | 1/4 |

The **odd-sized kernel** simplifies the virtual texture pixel shader a bit and has the nice property that texel positions with a unfiltered colour in a lower mip always exists in all higher mips. This allows specifying UV coordinates that get always get a defined colour in higher mip-maps. This is useful if you want to get seamless texture mapping or unique UV unwrapped geometry. The alternative is to add border pixels but that's only limiting the problem, not solving it.

Example: Gauss 3×3:

| | | |
|---|---|---|
| 1/16 | 2/16 | 1/16 |
| 2/16 | 4/16 | 2/16 |
| 1/16 | 2/16 | 1/16 |

Properties of the different mip-mapping kernel sizes:

| | Even mip-map kernel size | Odd mip-map kernel size |
|---|---|---|
| **Example kernels** | 2x2 box, 4x4 gauss, 4x4 sharpen | **3x3 gauss, 5x5 gauss sharpen** |
| **GPU creation support** | Yes (fast for a full mip chain) | **No but easy to implement in PS** |
| **GPU rendering support** | Yes (allow tri-linear and anisotropic) | **Bilinear yes, Mip mapped rendering has offset issues** |
| **Creation Speed** | fastest | **fast** |
| **Rectangular charts** | **filtering artefacts** | **Precise (texel center in higher mips remain at position)** |

Slightly better quality can be achieved by making use of the higher mip-maps (not only the next higher one). This is even more important if your mip-maps are not exactly half the size. As mentioned earlier it can be useful to keep the tile size including the border a power of two, but that requires a more complex mip-map generation algorithm.

**Hardware tri-linear filtering**

Just a note on trilinear filtering – trilinear filtering is simple to implement in the shader with additional texture fetches. If we wish to use hardware filtering for this, we need a mip-mapped tile cache, i.e. redundant storage and updates of the texture tiles. Additionally this requires a wider texture border and storing the tiles in two resolutions wastes memory and complicates the updating a fair amount.

## 2.3.4 Possible Tile Sources

### 2.3.4.1 *Streaming from Disk*

A very good source for virtual texture tiles can be the hard drive [WAVEREN06]. Requests are fulfilled in more or less constant time and memory requirements are constant. Normal IO functions on modern OS should be avoided because they try to cache and combine requests which not only results in variable latency but the memory for this cache is now competing with our application. Code or data which is not frequently accessed can be paged out and we get frame-rate hitching. Windows offers IO Completion ports which can be used efficiently and caching from the OS can be disabled. Using this you have a more direct hardware access but the reads now needs to align with the disk cluster size. As this can be different depending on which formatting the user chose, it's best to align your tile size and placement. The cluster size is a power of two and often in the range from 4K to 32k[13]. Knowing this it's clear that for best performance the header size of the streaming file should be 0 or padded to align with the cluster size. Each tile should be split in as few clusters as possible and preferably nearby to tiles that have a similar locality. With non static compression where each tile can have a different memory size this can be a bit tricky.

Streaming from DVD/CD/BD/HD DVD is more challenging as latency and bandwidth is much worse [NOGUCHI08]. On first sight the latency might not sound like a problem if you think of one tile being requested and coming in after that time. With very few tiles in flight you actually get good performance but with more tiles the seek time is dominating more and more. The alignment here is more important because the data is read and decoded in the cluster size and ignoring this is wasting the precious performance. It

[13] It's good to have the code hiding the cluster alignment from the user code so the code works for clusters of any size.

might make sense to consider a larger tile size to get better performance from hardware with high latency.

#### 2.3.4.2 *Streaming over Network*

Wouldn't it be nice to join a multiplayer match without the need to download a map? You can join an ongoing game and even see all the detailed changes like tire tracks and decals that happened to the map. The normal game server or some special extra server can bake decals and distribute them to the clients on demand. Compression might be more important here, but it seems that downstream bandwidth will be much less of a problem in the future.

#### 2.3.4.3 *Procedural Content Generation*

Nowadays especially massive multiplayer games require a huge amount of work to fill the world with details. Different areas should have a unique look to provide a richer gaming experience. Having a technique that allows rendering more detail doesn't help either.

Many techniques [BRUNETONNEYRET08] [LHN04] [ANDERSSON07] [WOODARD05] [QMK06] [GLANVILLE04] [DACHSBACHER06] are known to generate procedural content and if the generation step is fast enough this can be even done on the fly similar to a texture tile request from hard drive. Data can be generated on the CPU or the GPU. It's important to give the artists enough control over this; otherwise you only remove the tiled texture look without getting the content you want.

One of the simplest ways to generate huge texture resolutions is already used in many games rendering terrain; Terrain detail often consists of some tiled textures that are blended with interpolation information at a much lower resolution [BLOOM00]. This gives good results, especially when combined with a low resolution texture to add variation and break the tiling look.

*Crysis* uses terrain material blending (see *Figure* ) to get a detailed ground texture for a huge terrain but that method requires multiple materials to be blended at the pixel level. The overdraw can be up to three as each terrain vertex is assigned to one material and so blending within a triangle requires up to three passes. Additionally the detail materials are modulated by a low frequency texture.

Using such techniques in real-time gets slower the more features you pack into the system. Baking this in an offline process allows much more sophisticated blending and keeps the rendering performance constant. This also allows baking details like roads or tire tracks. You might have to use a lower resolution but this can be hidden with detail texturing.

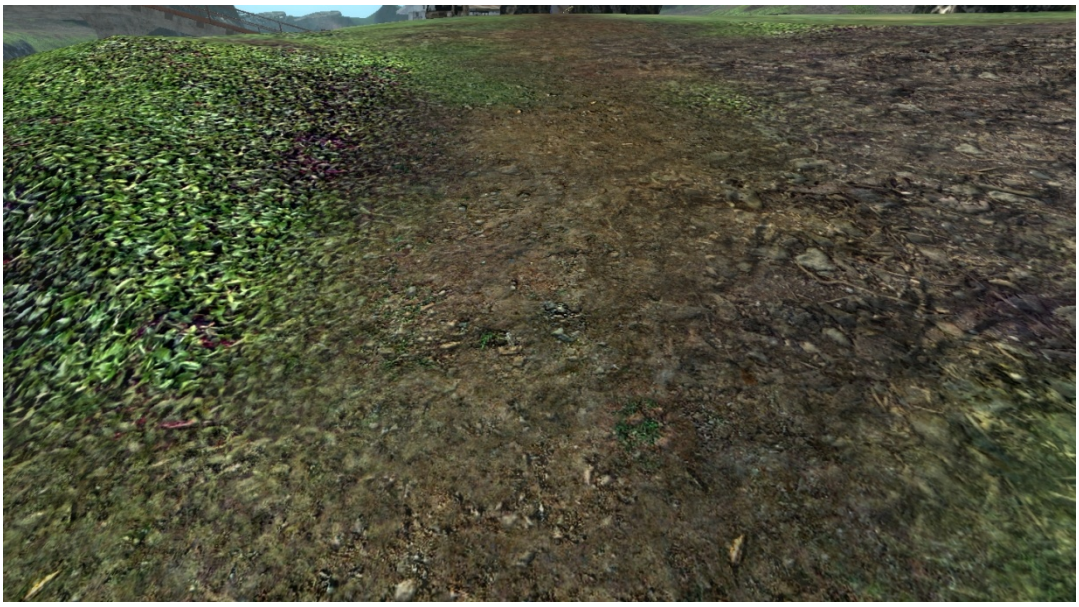

***Figure 5**. Terrain material blending as it can be seen in* Crysis *(terrain detail objects have been removed). Note that this is simply a reference – this particular screenshot was not implemented using virtual texture method*

Placing decals is one way to give the artists control over local areas. Decals can be used in multiply mode to bring brightness and colour variation, or they can be used in alpha-blend mode to locally replace existing content. As in nature, self similarity is common and it is smart to reuse the decal. This can be taken to the limit where a large number of placed decals form the surface itself. This allows breaking the tiled texture look but doing it without control results in a repetitive decal look.

***Figure 6**. Example of a scenario that would benefit from using virtual textures - decals in the game* Crysis *(roads, tire tracks, dirt) used on top of terrain material blending.*

*Decals* or *Roads* (as quantities in our game) can be seen as some special form of vector graphics which supports high resolutions naturally. Classic 2D vector graphics can be used for content like signs, advertisements or large scale paintings and even longer text sections are possible. Even large content like terrain with roads and other man made structure are suited for vector graphics content [BRUNETONNEYRET08]. Again the data can be generated on the fly or in an offline process.

The offline process has the advantage of constant rendering performance and even more important, constant memory requirements. Rendering 3D content is often no good idea as it requires additional memory for textures and meshes but there is one interesting application that doesn't suffer from this as it shares data with the normal scene rendering. [FFBG01] describes a method named „Adaptive shadow maps"(ASM) and for semi static scenes this can be an interesting option. The technique does not require any mesh unwrapping as the shadow map is projected from the light source perspective.

Other methods often require a unique UV unwrapping to store some intermediate data of the shading computation. Instead of storing the shadow map depth value, the light occlusion factor for a given surface position can be baked. Ambient occlusion, the effect of multiple lights or static global illumination can be baked as well. Taking this to the limit allows baking the final resulting colour of shading into a virtual texture. The data can still be dynamic as the tiles can be updated but now shading is decoupled from rendering and this brings along nice properties. It might be not the right time to ship games fully based on that concept as graphics hardware is not made for that. In real-time rendering this is known [BAKER05] but rarely used, in offline rendering decoupling shading from rasterization is used to great success in the REYES rendering system. In contrast to per-pixel-shading the REYES system does heavy tessellation of the geometric primitives, computes shading at that level and introduces other interesting things worth checking out [GUAN07].

### 2.3.4.5 *Sparse Textures*

Encoding material properties might be limiting and blending materials with alpha masks allows for much more flexible material blending. In [ANDERSSON07] sparse textures are used to compress this kind of data efficiently. Texture tiles with the same content can index to the same cache element and by using masks with larger areas of the same value this allows good compression. If only few tiles need to be stored you might not even require a dynamic cache. Then you just index with a static indirection texture into a static tile cache. This can be seen as a special case of the normal adaptive texture method and this way it integrates nicely with the dynamic virtual texture asset pipeline. Even multiple indirections for multiple masks can be stored in one indirection texture[14].

[14] As a hint: The UV position of the tiles in the texture cache can be encoded in one component and the scale might not be required.

### 2.3.4.6 *Combo Textures*

When using multiple masks the mentioned technique becomes less efficient. Compression of the sparse textures is only efficient when using one indirection per mask. An alternative is a novel technique we call the "Combo texture". The method allows to store up to $2^n$ masks in n texture channels in a compressed form. The compression is lossless under some controllable conditions and becomes lossy with complex material blends. Texture filtering can be used as usual but filtering introduces blending which again can suffer from the same compression issues.

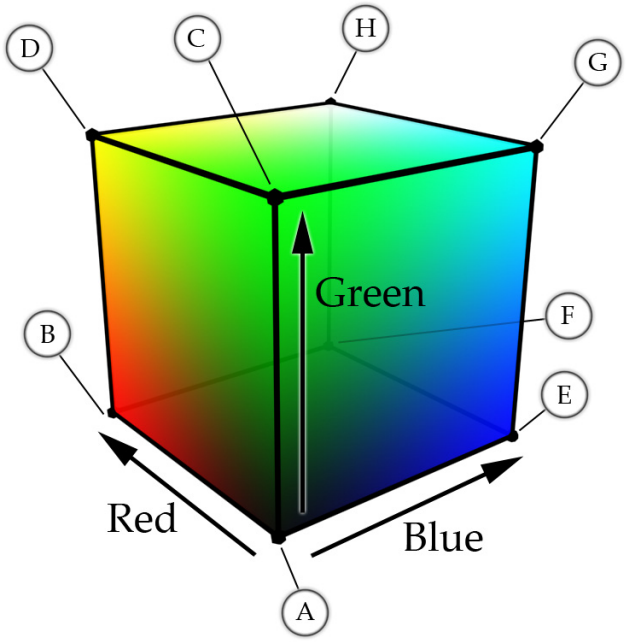


***Figure** 7.The RGB Cube shows how up to 8 materials (A..H) can be blended with one colour only.*

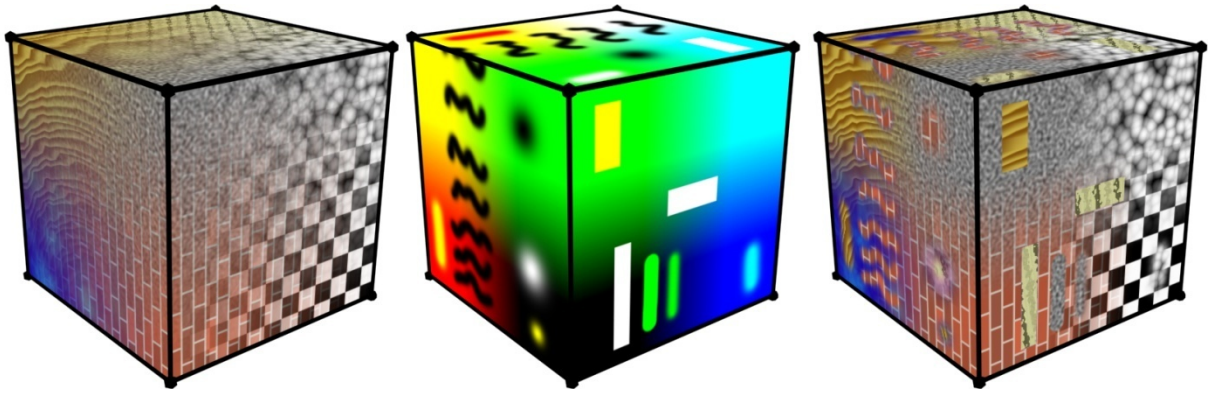

***Figure 8**. Multiple materials can be blended using the combo texture method.* Left*: Seven materials are blended on the RGB cube colours.* Middle*: A combo texture is projected on a cube, setup to blend materials in many different ways.* Right*: Seven materials are blended based on the combo texture from the image before.*

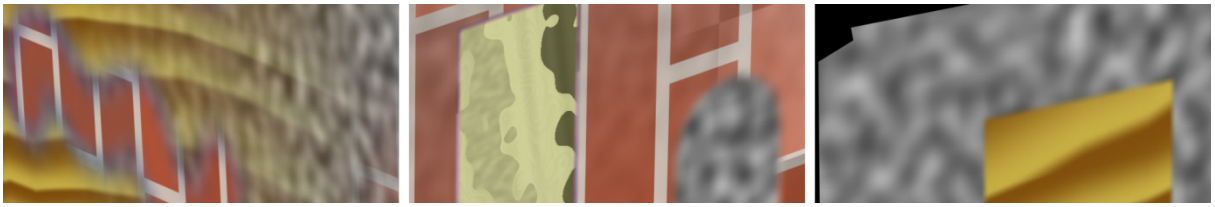

***Figure 9****.Combo texture results zoomed in*
*Left: Blending between two incompatible colours (yellow, black) causes leaking.*
*Middle: Bilinear filtering between two incompatible colours (black, white) causes leaking.*
*Right: Bilinear filtering between two compatible colours (green, yellow) blends nicely.*

If we use a 3 channel combo texture the properties of the method can be described on the RGB cube (*Figure* ). Each corner of the cube represents one material and eight corners allow therefore eight materials to be blended. All corners and the edges allow lossless compression as the linear blend between two materials can be expressed as a linear blend between two positions in the cube. Expressing arbitrary material mixes or blending between materials in general can introduce material-leaking[15] or the wrong amounts of certain materials. By smarty placing the materials in the right corners artefacts often can be avoided. Artists can paint these combo textures easily but if they paint material masks the material placement in the combo texture can be rearranged much easier at a later stage. Material masks can be converted to combo textures or any other representation before normal texture compression is applied in the art pipeline. If the material leaking becomes too apparent the bilinear filtering can be replaced by shader code that doesn't exhibit this problem. These artefacts are usually only visible in magnification where virtual textures can help to increase the resolution.

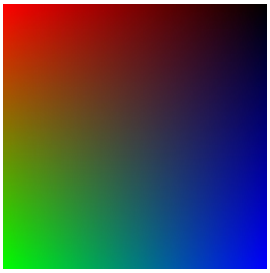

***Figure 10****. Four simple materials (colour: red, black, green, blue) are blended in a single pass which results in the best quality.*

The combo texture method can be implemented in a simple single pass shader if the shader of the different materials are simple and share common properties (e.g. detail materials, Phong materials that share textures and differ only in colours and specular power). *Figure* 10 shows four most simple materials (solid colour) blended in a single pass. With complex materials the possible shader permutation count explodes and a lot of bandwidth would be wasted by data that gets masked away.

[15] Materials that haven't been part of the mix are blending into the result.

A multi-pass solution based on frame-buffer blending would be more flexible.

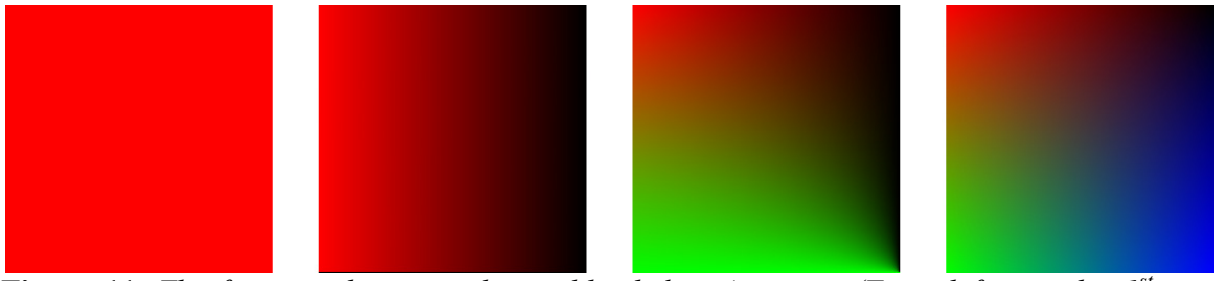

**Figure 11**. *The four simple materials are blended in 4 passes. (From left to right: 1st pass, 2nd pass, 3rd pass, final pass)*

Figure 11 demonstrates how such a technique works by using alpha blending. A simple implementation could use additive blending but this requires more than 8 bit precision in the frame buffer for an acceptable look. Better is to lerp[16] with the frame buffer content. To compute the blend factor we need to know the material blend factor and the sum of the blend factors we have blended already. Because frame buffer blending isn't flexible and precise enough we compute the sum in the shader. The following shader code illustrates the computation for eight materials:

```
float3 g_ComboMask;             // RGB material combo colour
                                // (3 channels for 8 materials)
                                //   000,100,010,110,001,101,011,111
float4 g_ComboSum0,g_ComboSum1; // RGBA sum of the masks blended so far
                                // including the current
                                // (8 channels for 8 materials)
                                //   10000000, 11000000, 11100000, 11110000
                                //   11111000, 11111100, 11111110, 11111111

float ComputeComboAlpha( BETWEENVERTEXANDPIXEL_Unified InOut )
{
  float3 cCombo = tex2D(Sampler_Combo,InOut.vBaseTexPos).rgb;

  float3 fSrcAlpha3 = g_ComboMask*cCombo + (1-g_ComboMask)*(1-cCombo);
  float fSrcAlpha3 = fSrcAlpha3.r*fSrcAlpha3.g*fSrcAlpha3.b;

  float4 vComboRG = float4(1-cCombo.r,cCombo.r,1-cCombo.r,cCombo.r)
                  * float4(1-cCombo.g,1-cCombo.g,cCombo.g,cCombo.g);

  float fSum = dot(vComboRG,g_ComboSum0)*(1-cCombo.b)
             + dot(vComboRG,g_ComboSum1)*(cCombo.b);

  // + small numbers to avoid DivByZero
  return (fSrcAlpha+0.00000001f)/(0.00000001f+fSum);
}
```

**Listing 4.** *Shader to compute the alpha value when rendering a material with the combo texture method*

[16] lerp = linear interpolation  A*(1-factor)+B*factor

Additional Notes:

- Using up to 8 materials and using an optional alpha channel for custom properties is nice and useful; when using more than three channels the technique becomes less intuitive and more artefacts can appear.
- A smart usage of combo textures might allow dropping other textures like diffuse or specular textures.
- You can modify the combo texture to dynamically blend between different materials (dirt, rust, damaged, wet).
- To avoid overdraw it is good to use alpha test and for static combo textures you can create a static index buffer per material.

### 2.3.5 Mesh Parameterization

The virtual texture method relies on unwrapped 3D models and that can be done with the usual techniques either by hand or in some automated way [FARIN02] [CARRHART02]. Knowing that the unwrapping is used with a virtual texture the layout can be optimized for better performance. Unused areas are common in all unwrapping methods but packing the data specifically to reduce the amount of wasted texture tiles allows faster loading with less cache and storage memory required. The packing algorithm needs to avoid the quad-tree borders at multiple levels (see Figure 12). Another important criterion is to aim for similar locality between the texture space and world space, that again at the granularity of the texture tiles.

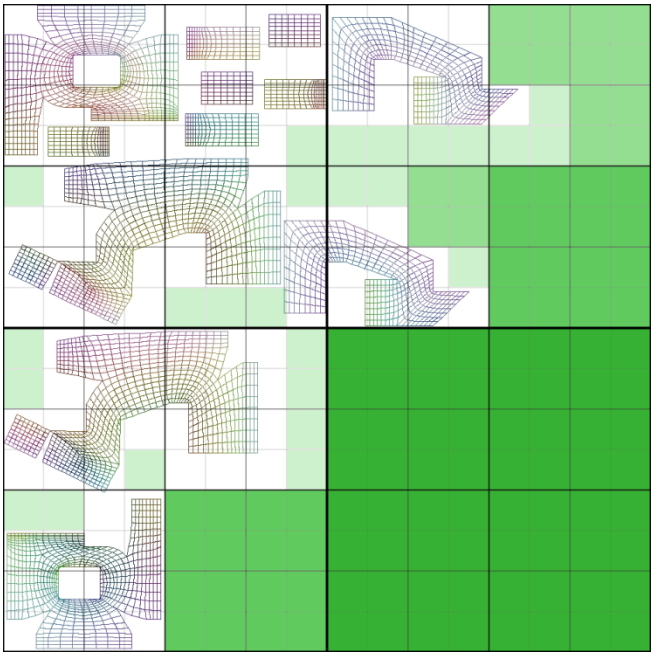

**Figure 1**. *By carefully placing the unwrapped mesh in the UV space less tiles are requires at multiple levels of the quad tree (green tiles don't need to be stored).*

#### 2.3.5.1 *Unique Unwrapping*[17]

The unwrapping does not have to be unique when using virtual textures but having a unique unwrapping can have additional benefits. For the virtual texturing a unique mapping has a higher but more consistent memory demand in the texture cache (see "Computing the local LOD required"). Dynamic tile content virtually requires unique unwrapping as texture updates should not affect other areas. Even static content might require unique unwrapping if the content cannot be shared. Examples for that are light maps or object-space normal maps. Normal maps in tangent-space are easier to compress but object/world-space allows better quality and simpler shaders.

A simple form of unique unwrapping almost unwraps every geometric primitive (triangle, quad, grid, …) individually. Once tessellation is commonly supported in hardware this together with displacement maps or geometry images might be a good solution [THC*04] [RECMULTI4E][LMH00]. It should be mentioned that this would allow moving the indirection to the primitive to become more efficient.

### 2.3.6 Computing the Local LOD

#### 2.3.6.1 *How Much Tile Cache Memory is Required?*

As mentioned in "mesh parameterization" when using a unique mapping the virtual texturing method has a higher but more consistent memory demand in the texture cache.

This is because non unique mappings allow sharing some tiles if the viewer is close to a 3D surface that shares the same area on the virtual texture. Assuming unique unwrapping the memory required in the texture cache can be approximated by multiplying the screen width and height with the average over-draw and a user defined quality factor. The computation is only approximating the real value because it ignores mip-mapping, anisotropy, bilinear filtering and waste because of tile borders. The wasted memory becomes more when used with a bigger tile size as full tiles are required even if only partly used. The pixel overdraw is 0 if that pixel is not using the adaptive texture method and it's greater than 1 if alpha blending is used.

Graphic cards normally compute the mip-map level depending of the screen texture derivation in *x* and *y* for a two pixel block[18]. For high anisotropy levels like for walls or roads seen in the distance the mip-level for *x* and *y* can be quite different. Non anisotropic filtered lookup on the graphic card here uses the maximum of the values

[17] Unique texture unwrapping: A position on the texture is only mapped to one model position; there is no reuse of the texture.

[18] ddx() is computed from an odd/even horizontal pixels pair, ddy() from a vertical pair

and as a result the texture looks blurry but it benefits from good texture cache coherence and has no shimmering in motion.

***Figure 2.*** *Different texture filtering techniques shown on rendering of a road section*

In *Figure* a road is shown with different texture filtering modes. Starting from the left you can see standard bilinear which only picks the lowest mip-map. The second image shows the bilinear filtering without the use of mip-maps. That looks good on the screenshot but in motion the noisy aliasing is very apparent. The following images show 2X, 4X and 8X anisotropic texture filtering. Note how the white stripes keep the details in the distance depending on the anisotropic filtering level.

#### 2.3.6.2 *Computing Exact Local Tile LOD*

With a high frame-rate on current hardware it's usually not possible to get the local LOD and the resulting update accomplished within the same frame. Then the perfect local LOD would have to include future frames and that's almost unsolvable for dynamic scenes. We have to assume a latency of multiple frames and therefore it's better to find a more conservative LOD computation.

Occluded tiles can save tile cache space but those might become quickly visible. To find the texture tiles that are hidden behind objects for many frames a high level streaming prediction system based on some occlusion system (e.g. a static PVS) can help.

Rapid view direction changes are common in real-time games and it is important to integrate in the local LOD computation.

#### 2.3.6.3 *Approximating Local Tile LOD*

A reasonably conservative approximation can be acceptable and can sometimes even be better than the exact LOD for the current frame. It's better to have tiles already available to be prepared for quick camera angle changes or fast object movements. Computing the LOD based on the distance per triangle or draw call can be such an approximation (easy to implement for height-maps[19]). For content that has a varying world to Texel density the LOD computation can be extended further.

[19] Height-map rendering can make use of the virtual texture cache without any pixel shader indirection if the mesh is organized in a similar quad-tree structure.

#### 2.3.6.4 Exact Local Tile LOD in View Space with Occlusion

In [BARRETT08] the author describes a method to compute the local LOD by rendering the view as a pre-pass with some special shader that outputs the tile-id with the required mip-map level. The resulting image is used to get the LOD required for some local area of the virtual texture. It can be tricky to get the data back to the CPU efficiently and some latency cannot be avoided. With a high frame-rate this can be multiple frames and therefore the cause of artefacts. Hidden object parts that become visible (from occlusion, animation or by entering the view) suffer from temporal local blurriness.

#### 2.3.6.5 Exact Local Tile LOD in Texture Space

Most of the mentioned problems can be avoided when computations are performed in texture space [LDN04B]. Such methods even work outside of the view or when being occluded by other objects or the object itself. The object needs to be rendered in texture space using a medium resolution with a pixel shader computing the colour of the Texel as LOD for the point in world space. Overlapping primitives in the texture space require extra treatment which is of course not required when using a unique UV unwrapping. The overlapping areas should get the colour value of the highest LOD required there and luckily the z buffer can easily be abused for that.

Then the data can be extracted for different texture areas with Occlusion queries or other methods like CPU read-back. This 2 pass algorithm allows multiple tests in different areas of the quad-tree. As the returned information is changing slowly the renderings can be distributed over multiple frames without introducing severe problems.

We haven't tried the GPU method; instead we implemented a CPU based method. It was easier to implement (with some approximations) and doesn't suffer from the latency issues the GPU solution has. The idea here is to build up a quad tree in texture space to compute distances to quad-tree nodes in world space. That can be CPU heavy and memory intensive but it's a good reference solution.

The described methods have quite some different properties and it depends on the application and hardware platform which works best. The view space method with occlusion only returns the minimal set of tiles required but it offers no good prediction. Concerning prediction, the other two methods show their strengths and a combination with one of them can be a good idea. Which one is the better choice depends on the platform and on the data. Computing the local tile LOD per triangle is very bad with huge triangles spanning multiple tiles. This can be solved by subdividing the mesh but then the method becomes more complex and memory intensive.

## 2.4 Future Directions

Many interesting graphic algorithms rely on some form of LOD query and local texture updates but the currently available hardware and API (DirectX®/OpenGL) is not supporting this very well.

We [game developers] need asynchronous partial texture update and mip-level adjustment with predictable performance. It seems we get virtualized memory for graphic card memory which is nice from an OS perspective but for games we don't want stalling virtual memory. The application or the engine can deal with missing data on a more high level and can provide fallbacks until the request is resolved. Whatever methods are used and the described virtual texture method is just one, we need to be able to deliver frame rate quality and see this as a valuable feature (Quality of Service).

## 2.5 Acknowledgements

This chapter wouldn't be the same without the passionate work of the many programmers, artist and designers at Crytek. Working there is both inspiration and demanding as we take every good idea to the limit and challenge it against other methods. Thanks to Efgeni Malachewitsch for the 3D model, Nick Kasyan, Anton Kaplanyan, Michael Kopietz and all others that helped me with this text. Special thanks to Natasha Tatarchuk, Kev Gee, Miguel Sainz, Yury Uralsky, Henry Morton, Carsten Dachsbacher and the many others from the industry for the interesting discussions on my favorite topic: Graphics

## 2.6 References

[ANDERSSON07] ANDERSSON, J. 2007. Terrain Rendering in Frostbite using Procedural Shader Splatting, Advanced Real-Time Rendering in 3D Graphics and Game, Course 28, Siggraph 2007, San Diego, CA. http://ati.amd.com/developer/gdc/2007/Andersson-TerrainRendering(Siggraph07).pdf

[BAKER05] BAKER, D. 2005. Advanced Lighting Techniques, Meltdown, Seattle 2005. http://www.slideshare.net/mobius.cn/advanced-lighting-techniques-dan-baker-meltdown-2005

[BARRETT08] BARRETT, S. 2008. Sparse Virtual Texture Memory, Game Developer Conference, San Francisco, CA. http://silverspaceship.com/src/svt

[BLOOM00] BLOOM, C. 2000. Terrain Texture Compositing by Blending in the Frame-Buffer, http://cbloom.com/3d/techdocs/splatting.txt

[BRUNETONNEYRET08] BRUNETON, E.AND NEYRET, F. 2008. Real-time rendering and editing of vector-based terrains, In Proceedings of Eurographics 2008, Vol. 27, Num. 2, http://www-evasion.imag.fr/Publications/2008/BN08/article.pdf

[CARRHART02] CARR, N. A. AND HART, J. C. 2002. Meshed Atlases for Real-Time Procedural Solid Texturing, ACM Transactions on Graphics (TOG), , Vol. 21, No. 2, pp. 106 – 131, http://graphics.cs.uiuc.edu/~nacarr/papers/rtpst.pdf

[DACHSBACHER06] DACHSBACHER, C. 2006.Cached Procedural Textures for Terrain Rendering, ShaderX$^4$: Advanced Rendering Techniques, Engel, W. (Editor), Charles River Media, Cambridge, MA.

[FFBG01] FERNANDO, R., FERNANDEZ, S., BALA, K., AND GREENBERG, D. P., 2001. Adaptive Shadow Maps. In Proceedings of ACM SIGGRAPH 2001, Computer Graphics Proceedings, Annual Conference Series, pages 387–390

[GUAN07] GUAN, S.-H. 2007. Reyes and Shader Pipeline, http://www.scribd.com/doc/7346/Reyes-and-Shader-Pipeline

[GLANVILLE04] GLANVILLE, R. S.2004. Texture Bombing, in *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, Fernando, R. (Editor), Addison-Wesley, April 2004. http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch20.html

[QMK06] Qin, Z., McCool, M. D. AND Kaplan, C. S. 2006. Real-Time Texture-Mapped Vector Glyphs, I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games, pp. 125 – 132, Redwood City, CA. http://www.cgl.uwaterloo.ca/~zqin/i3d2006/ft_gateway.cfm

[LLOYD05] LLOYD, B., YOON, S., TUFT, D. AND MANOCHA, D. 2005. Subdivided Shadow Maps, Technical Report TR05-024, University of North Carolina at Chapel Hill, http://gamma.cs.unc.edu/ssm/ssm_TR05-024.pdf

[LEFOHN05] LEFOHN, A.2005. A Dynamic Adaptive Multi-resolution GPU Data Structure, GPGPU: general-purpose computation on graphics hardware, Siggraph 2005 Courses, Los Angeles, CA, August 2005, http://www.gpgpu.org/s2005/slides/lefohn.AdaptiveCaseStudy.ppt

[FARIN02] FARIN, G., FEMIANI, J., AND RAZDAN, A. 2002. Parametrizing Triangulated Meshes, Seminar, PRISM RA, 2002. http://prism.asu.edu/publications/pr_list_details.php?ref_num=117

[IDTECH507] id Software Debuts id Tech 5 Press Release, 2007 http://www.idsoftware.com/business/press/index.php?date=20070611000000

[NVIDIA04] NVIDIA WHITEPAPER, 2004. Improve Batching Using Texture Atlases http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/BatchingViaTextureAtlases/AtlasCreationTool/Docs/Batching_Via_Texture_Atlases.pdf

[EPHANOV06] EPHANOV, A. AND COLEMAN, C. 2006. Virtual Texture: A Large Area Raster Resource for the GPU, http://www.multigen-paradigm.com/pdf_content/2006IITSEC_VTPaper_2.pdf

[LMH00] LEE, A., MORETON, H. AND HOPPE, H. 2000. Displaced subdivision surfaces, In Proceedings of SIGGRAPH 2000, pp. 85-94, August 2000. http://research.microsoft.com/~hoppe

[LHN04] LEFEBVRE, S., HORNUS, S. AND NEYRET, F. 2004. All-Purpose Texture Sprites, ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5209.pdf

[LDN04B] LEFEBVRE, S., DARBON, J. AND NEYRET, F., 2004. Unified Texture Management for Arbitrary Meshes, Technical Report RR5210-, INRIA, Number RR5210, May 2004, http://www-evasion.imag.fr/Publications/2004/LDN04/RR-5210.pdf

[LEFOHN03] LEFOHN, A. 2003. Dynamic Volume Computation and Visualization on the GPU, http://www.vis.uni-stuttgart.de/vis03_tutorial/lefohn.pdf

[NOGUCHI08] NOGUCHI, M. 2008. Running Halo 3 Without a Hard Drive, Presentation http://www.bungie.net/images/Inside/publications/presentations/Loading_done_gdc_2008.pptx

[THC*04] TARINI, M. HORMANN, K., CIGNONI, P. AND MONTANI, C. 2004. PolyCube-Maps, In Proceedings of Siggraph 2004, pp. 853-860, Los Angeles, CA, 2004. http://vcg.isti.cnr.it/polycubemaps/resources/sigg04.pdf

[WAVEREN06B] J.M.P. VAN WAVEREN, 2006. Real-Time Texture Streaming & Decompression, http://softwarecommunity.intel.com/articles/eng/1221.htm

[WOODARD05] WOODARD, T. 2005. Real-time GPU-based texture Synthesis, In proceedings of IMAGE 2005, http://www.diamondvisionics.com/docs/Real-time%20GPU-based%20Texture%20Synthesis.pdf

Chapter 3

# March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatures on GPU

Jeremy Shopf[4] Joshua Barczak[5] Christopher Oat[6] Natalya Tatarchuk[7]

Game Computing Applications Group
AMD, Inc.

***Figure 1.*** *Froblins navigate to a mushroom patch goal and harvest food.*

[4] jeremy.shopf@amd.com
[5] josh.barczak@amd.com
[6] chris.oat@amd.com
[7] natalya.tatarchuk@amd.com

## 3.1 Beyond Pretty Pictures toward Intelligent Interactive Experiences

Artificial intelligence (AI) is generally considered to be one of the key components of a computer game. Sometimes when we play a game, we may wish that the computer opponents were written better. At those times while playing against the computer, we feel that the game is unbalanced. Perhaps the computer player has been given different set of rules, or uses the same rules, but has more resources (health, weapons, etc.). The complexity of underlying AI systems, along with game design, belies the resulting feeling we have when playing any game. As the CPU and GPU speed and power continues to grow, along with increasing memory amounts and bandwidth, game developers are constantly improving the graphics of their games. In the last five years the production quality of games has been increasing (along with the corresponding budgets). Recent games woo players with incredible breakthroughs in real-time 3D graphics, complexity of the worlds and characters, as well as various post-processing effects. And while there had been tremendous improvements for parallelizing rendering through the evolution of consumer GPU pipelines, artificial intelligence computations are treading behind. To date, there had been rather few attempts at parallelizing AI computations.

Typically, in a game, AI controls the behavior of non-player-characters (NPC), whether they are friendly to the player or act as game opponents. This may include actual characters, or it can simply be tanks and armies (such as in a real-time strategy game), or monsters in a first-person shooter. The uniform feeling is the better the AI is, the better the game. A more sophisticated AI system allows for more interesting and fun gameplay. Artificial intelligence is used for various parts of the game. Typical computations include path finding, obstacle avoidance, and decisions making. These calculations are needed regardless of the genre of interactive entertainment, be it a real-time strategy game, an MMORPG, or a first-person shooter. We will also soon see dynamic character-centric entertainment in the form of interactive movies, where the viewer will have control over the outcome.

In many scenarios, the AI computations include dynamic path finding. This involves auto-simulating characters' behavior, and/or running a terrain analysis to identify good or update valid paths as result of gameplay. These computations can be quite a hog on CPU time budget, even in multi-core scenarios. As a result, many game developers are looking for ways to minimize the CPU hit of pathfinding. Because path finding and AI in general is such a compute-intensive, expensive calculation, we often see boring, zombie-like NPC interaction. Furthermore, when gameplay and physics are simulated on the CPU, and the characters are rendered on the GPU, there is an additional PCI-E data transfer overhead for character positions and state. If we can utilize the GPU for running in-game AI code not only we can speed up path finding, but we can also introduce a number of other interesting effects. Our characters can start living on their own, resulting in so-called "emergent behaviors" – such as lane formation, queuing, and reactions to other characters and so on. And this means that game play will be a lot more fun!

With this in mind, we set out to explore the next visual frontier for interactive experience, combining massively parallel AI computations with high fidelity rendering algorithms. In this chapter we will describe the simulation and rendering methods used for the AMD *Froblins* demo, designed to showcase many of the new approaches for character-centric entertainment. These techniques are made possible by the massively parallel compute available on the latest commodity GPUs, such as ATI Radeon® HD 4800 series. In our fantasy large-scale environment with thousands of highly detailed, intelligent characters, the

*Froblins* (frog goblins), are concurrently simulated, animated and rendered entirely on the GPU. The individual character logic for each froblin creature is controlled via a complex shader (over 3200 shader instructions). We are utilizing the latest functionality available with the DirectX® 10.1 API, hardware tessellation, high fidelity rendering with 4X MSAA settings, at HD resolution with gamma-correct rendering, full high dynamic range FP16 pipeline and advanced post-processing effects. The crowd behavior simulation is performed directly on the GPU. We will describe a GPU-friendly path-planning framework for large-scale crowd simulation. This framework can also be used to simulate larger crowds of simplified agents with smaller polygonal count. Our system has been used to simulate 65,000 agents at real-time frame rates on a single commodity GPU. By combining a continuum-based global path planner with a fine-grained agent-based local avoidance model, we can perform expensive global planning at a coarse resolution and lower update rate while the local model takes care of avoiding other agents and nearby obstacles at a higher frequency. To our knowledge, this is the first massive crowd simulation performed entirely on a GPU.

In our interactive environment we render thousands of animated intelligent characters from a variety of viewpoints ranging from extreme close-ups (with individual characters rendered at over 1.6 million triangles for close-up detail) to far away "bird's eye" views of the entire system (over three thousands characters at the same time). Our system combines state-of-the-art parallel artificial intelligence computation for dynamic pathfinding and local avoidance on the GPU, massive crowd rendering with LOD management with high end rendering capabilities such as GPU tessellation for high quality close-ups and stable performance, terrain system, cascaded shadows for large-range environments, and an advanced global illumination system. We are able to render our world at interactive rates (over 20 fps on ATI Radeon® HD 4870) with staggering polygon count (6 – 8 million triangles on average at 20-25 fps), while maintaining the full high quality lighting and shadowing solution.

## 3.2 Artificial Intelligence on GPU for Dynamic Pathfinding

### 3.2.1 Global Pathfinding

Many systems for crowd simulations rely on agent-based solutions, where the movement is computed for individual agents separately. While there are certain advantages to this approach (independent decisions for each agent, individual visibility and environment information), it is also challenging to develop behavioral rules for the agents that result in a consistent and realistic overall crowd movement. At the same time, scaling agent-based methods for a large number of agents is prohibitively computationally expensive, which is a concern for interactive scenarios, such as video games.

For our crowd simulation solution we chose a continuum-based approach similar to the *Continuum Crowds* work by Treuille et al. [TCP06]. This method converts motion planning into an optimization problem, using well-known numerical methods from optics and general physics for stable navigation solution.

This type of method is particularly well suited for simulating large numbers of agents because it is computed spatially, instead of per-agent, and results in smooth movement with no "dead-ends". Additionally, a continuum approach results in flow-like movement which is characteristic of actual large

crowds. The global model is only an approximation to accurate long-term planning with full visibility and decision logic, and therefore it is augmented by local collision avoidance problem. Together these methods produce smooth and realistic crowd movement, especially in areas of dense congestion.

In this continuum-based crowd simulation, the environment is formulated as a cost function (sometimes referred to as a speed function). This cost function incorporates both the achievable speed (based on terrain slope, etc.) and avoidance factor (based on agent density, large-scale obstacles, etc.) for locations in the environment. This cost function describes the travel-time to move from one location to a neighboring location and is used to evaluate the optimal path.

This cost function is then used as input to a solver that calculates the total travel-time (potential) from any location to the nearest goal. This potential ($\varphi$) is calculated such that it satisfies the *eikonal equation*:

$$\|\nabla\varphi(x)\| = F, \qquad (1)$$

where $F$ is the positive-valued cost function evaluated in the direction of the gradient $\nabla\varphi(x)$. It is intuitive to see how the function $\varphi$ could be constructed by integrating the cost function along the shortest-path from every location $x$. In this context, we can think of the global crowd movement as computing a propagating wave front, following the path of least resistance.

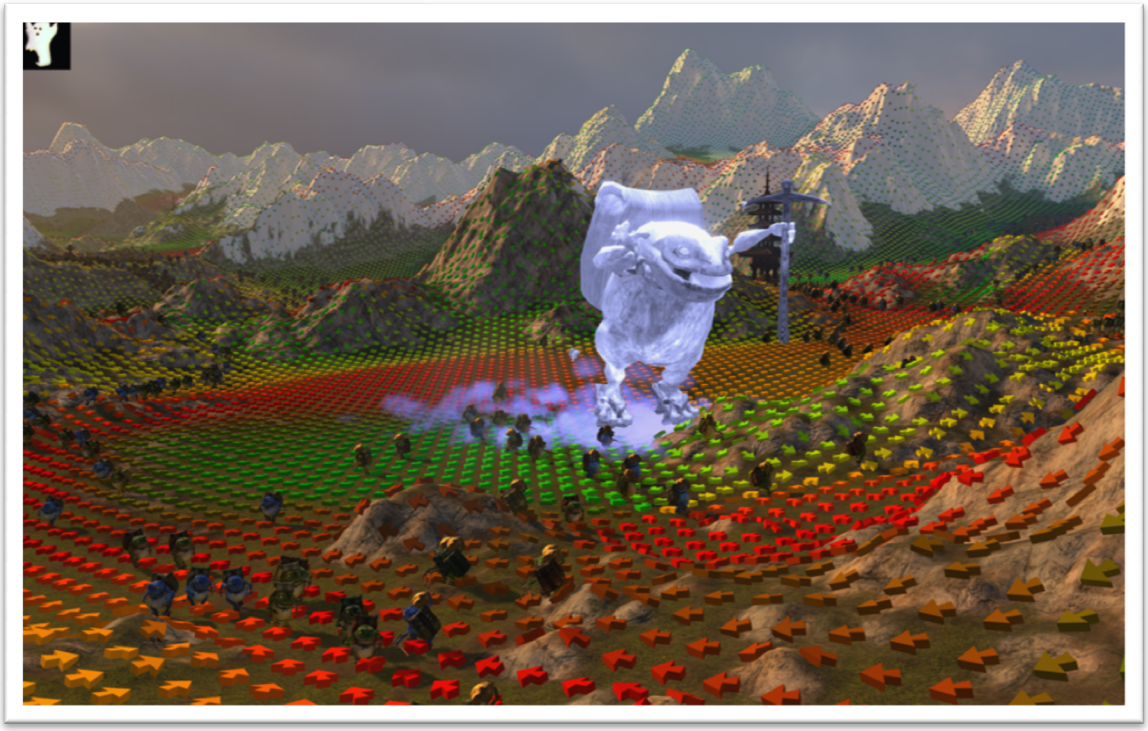

***Figure 2.*** *Example of dynamic path finding in game-like scenario. The arrows visualize character movement directions. As the large ghost Froblin scares away the little critters, they scamper away. Note that the arrows near the "monster" are pointing away, directing the characters away from a potential threat.*

By following the gradient of the generated potential field, agents are guaranteed to always be moving along the shortest path to the global goal considering the speed at which an agent can travel based on terrain features, obstacles and agent density (congestion).

#### 3.2.1.1 *Global Pathfinding CPU Implementation*

A fast and simple-to-understand computational algorithm to approximate the solution to the eikonal equation is the *Fast Marching Method* [TSITSIKLIS95], which we will summarize here. Because the potential is only known for the goal location, we begin by setting $\varphi = 0$ at the goal cell and adding this cell to a list of `KNOWN` cells. All other cells are added to an `UNKNOWN` list, with their potential set to $\infty$. The algorithm then proceeds as follows:

1) All `UNKNOWN` cells adjacent to a `KNOWN` cell are added to a `NEIGHBOR` list
2) The potential at each `NEIGHBOR` cell is calculated based on the potential at the neighboring `KNOWN` cells and the cost to get from the `KNOWN` cells to the `NEIGHBOR` cell in question
3) Update the `NEIGHBOR` cell with the smallest potential found in step 2 and add it to the list of `KNOWN` cells
4) Repeat until all cells are in the `KNOWN` list

Note that the above algorithm is identical to Dijkstra's method. The difference between Dijkstra's and the Fast Marching Method is the way that the potential is calculated in step 2. Solving the continuous eikonal equation by using Dijkstra's method on a discrete grid will not converge; we will always get stair-stepping artifacts regardless of the number of times you refine the grid structure.

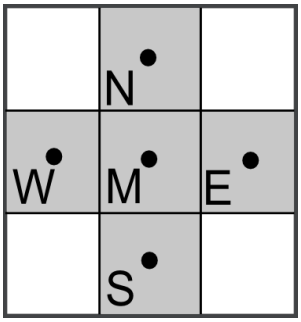


***Figure 3.*** *Illustration of neighboring cells used in finite difference approximation.*

Tsitsiklis presents a finite difference approximation to the continuous eikonal equation that eliminates the stair-stepping problem. First, the *upwind directions* are identified as the least costly neighbors in the x and y directions (see also Figure 3):

$$n_x = \operatorname{argmin}_{i\epsilon\{W,E\}}\{\varphi_i + C_M\} \qquad n_y = \operatorname{argmin}_{i\epsilon\{N,S\}}\{\varphi_i + C_M\} \qquad (2)$$

The finite difference approximation is then computed using the greatest solution to $\varphi_M$ in the quadratic equation:

$$\left(\frac{\varphi_M - \varphi_{n_x}}{C_M}\right)^2 + \left(\frac{\varphi_M - \varphi_{n_y}}{C_M}\right)^2 = 1 \qquad (3)$$

In the case that $n_x$or $n_y$ is undefined (neither neighbor along an axis is KNOWN), then eliminate the term containing that axis from the equation. Once $\varphi_M$is found for all cells, the gradient $\nabla\varphi$ can be easily calculated.

However, the *Fast Marching Method* is a serial algorithm and not amenable to parallelization, and, by extension, not highly suitable for efficient GPU computation.

#### 3.2.1.2 *Global Pathfinding GPU Implementation*

Luckily, there exists a method for solving the eikonal equation in parallel. The *Fast Iterative Method* [JEONGWHITAKER07A; JEONGWHITAKER07B] uses the same upwind finite difference approximation described in Section 3.2.1.1 but requires no ordered data structures to maintain lists such as KNOWN, UNKOWN, etc. The idea is to only perform updates to the potential function at the band of cells which are *active*. In practice, a list of individual active cells does not need to be maintained. A list of active *tiles*, or spatially coherent blocks of cells, is maintained. Intuitively, the list of active tiles is initialized to contain the tile containing the source (the goal in our application). We summarize the algorithm as follows:

1) Iterate *n* times on all cells in the active tiles;
2) Compare each cell in each active tile to the previously computed potential value for that cell. If the difference is within some small threshold, mark it as converged ;
3) For each active tile, perform a *reduction* on the convergence results to determine if the entire tile is converged;
4) Perform one iteration on all tiles neighboring the tiles determined to have converged in step 3 to see if any cell values change;
5) Update the active list of tiles to reflect all tiles that became inactive due to convergence or that were identified as being reactivated in step 4.

The authors reported 4-6X performance improvements over optimized CPU implementations for their tile-based implementation.

However, for our implementation, we are able to further simplify this algorithm because the complexity of our cost function does not vary greatly and our cell grids are small relative to the large datasets used in the authors' work.

Because our datasets are small ($128^2$ or $256^2$), the constant overhead of performing tests for convergence outweighs the gains from culling computation and impacts performance negatively. In other aspects, our algorithm is similar to the above. In order to ensure that our solver converges, we need to be able to make a conservative estimate of the number of iterations needed. The number of iterations used in our solver was determined empirically by examining worst-case cost function complexity.

By calculating four eikonal solutions at once, we are able to achieve 98% ALU utilization on an ATI Radeon™ HD 4870 with GDDR5 memory. This yields very high computational throughput. Our GPU based solver computes a $256^2$ solution in 20 ms which is faster than our CPU implementation by a factor of approximately 45. The performance data was collected on an AMD Phenom™ X4 Quad-Core CPU system

with 2GB of RAM and an ATI Radeon™ 4870 graphics card with 512MB of GDDR5 video memory and regular engine and memory clocks. HLSL source code for our iterative eikonal solver is listed in Appendix A.

#### 3.2.1.3 *Constructing the Cost Function*

Solving the eikonal equation requires a cost function that can be evaluated at each grid cell. The cost function for the environment in the *Froblins* demo is computed as follows:

$$F(x) = aT(x) + bD(x) + cA(x), \qquad (4)$$

where $F$ is the final cost function, $T$ is the static movement cost (including terrain movement cost equivalent to slope as well as any large static objects such as buildings), $D$ is the density of agents and $A$ is the cost related to dynamic hazards. $a$, $b$, and $c$ are weights that can be adjusted spatially to encourage different pathing depending on the situation. For example, it may be desired to increase the cost due to agent density near a goal to prevent overcrowding at goals.

Once the scalar cost function $F(x)$ is constructed (Figure 4), it can be supplied to the eikonal solver to calculate $\varphi(x)$. The gradient of $\varphi(x)$ is calculated using central differences.

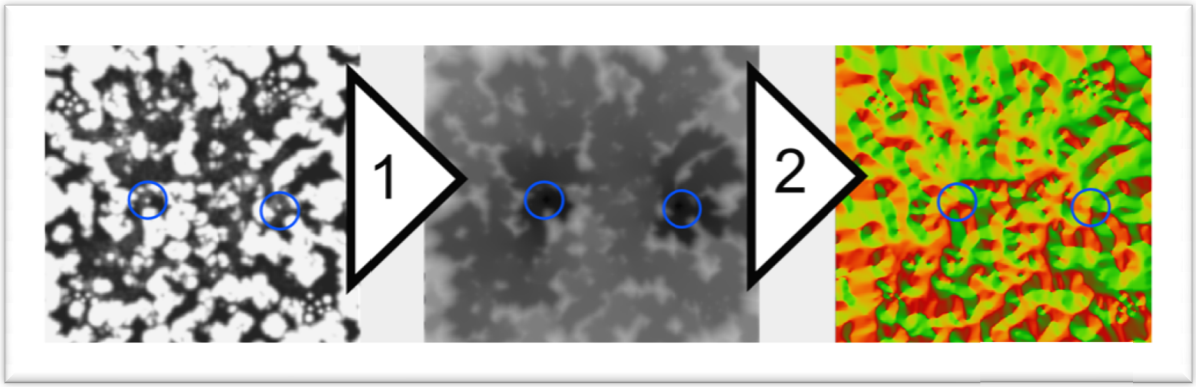


***Figure 4.*** *Left to Right: Cost function, potential, gradient of potential. The goals (sources) are at the center of the blue marker.*

### 3.2.2 Local Navigation and Avoidance

Unfortunately, solving the eikonal equation at a resolution high enough for large numbers of agents to avoid each other with acceptable fidelity is prohibitively expensive for a real-time application. In order to have an accurate behavior model for our characters, we augment our global eikonal solution with a local avoidance model that resolves these fine-grained obstacles.

The basic goal of a local model is to provide each individual agent with a velocity that will prevent collisions with nearby agents and also to navigate around obstacles and agents towards its desired

destination. This is typically handled by a continuous cycle of examining the nearby environment and reacting based on the discovered information.

#### 3.2.2.1 *Method*

Our local navigation and avoidance model computes agent velocities by examining the movement direction determined by the global model and the positions and velocities of nearby agents. This avoidance model is based on the *Velocity Obstacle* formulation [FIORINISHILLER98; vBPS*08].

For our model, we present each agent as a disc. Each agent $A_i$ therefore has a position $\mathbf{p}_i$, a velocity $\mathbf{v}_i$, a radius $r$, a maximum speed $s_i$, and a global goal direction $\mathbf{g}_i$ provided by the global solver. We infer an orientation $\theta_i$ from $\mathbf{v}_i$ by assuming that the agent is oriented towards $\mathbf{v}_i$. In our application, all agents have a similar radius and therefore $r$ is constant for all agents.

As in most local models, updating $\mathbf{p}_i$ and $\mathbf{v}_i$ for $A_i$ requires knowledge of the $\mathbf{p}_n$ and $\mathbf{v}_n$ for all agents $A_n \in A_{near}$, where $A_{near}$ is the set of all agents within a certain distance.

#### 3.2.2.2 *Spatial Queries via Novel GPU Binning*

Determining the positions and velocities of dynamic local obstacles requires a spatial data structure containing all obstacle information in the simulation. We developed a novel multi-pass algorithm for sorting agents into spatial bins directly on GPU. Our algorithm uses a 2D depth texture array and a single 2D color buffer to construct a data structure for storing agents in bins. The depth texture array serves as our *Agent ID Array.* A given 2D texel address in this array serves as a bin. A single bin is a 1D array texture array slice. The bin array grows down through successive texture array slices. Each slice of the texture array contains a single agent ID (bin element). The agent IDs are stored in bins in ascending sorted order by agent ID. The number of agents that fall into a given bin may be less than the bin capacity (which is defined by the number of depth array slices). In order to efficiently query the agent IDs in a given bin we use a *Bin Counter.* The Bin Counter is a 2D color buffer that records the load on each bin in the Agent ID Array.

To find all agents near a particular world space position, the position is translated into a 2D bin address. Any translation function may be used. Our world domain is square so a simple uniform grid was used to map world space positions to bins. Once the bin address is known, the bin load is read from the Bin Counter. This gives us the number of agents that are in the bin we are interested in. Finally, the each agent's ID in the bin is read from the Agent ID array.

This data structure must be updated each frame as agents move about the world. Updates are performed using an iterative algorithm that begins with all agent IDs in a buffer called *the working set.* Each iteration, as agents are placed into bins, they are removed from the working set. The algorithm continues to iterate until the working set is reduced to zero.

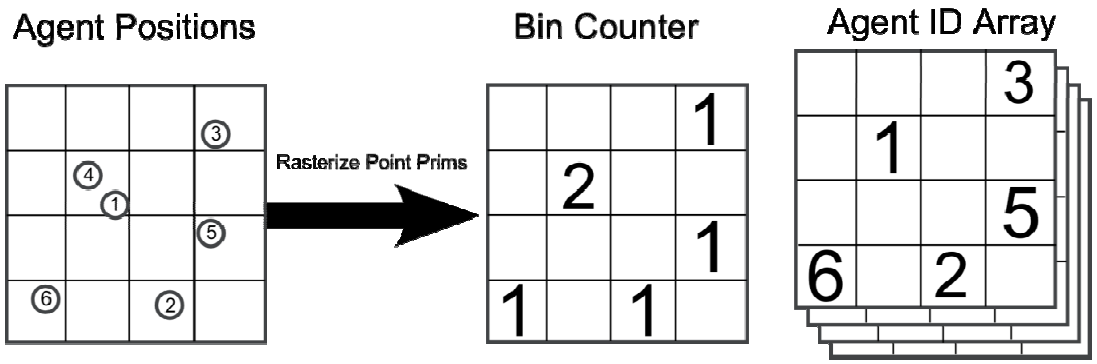


***Figure 5.*** *Agent positions are rasterized into a bin counter and an array containing IDs of agents in that bin. These IDs are later used to retrieve agent positions and velocities.*

We begin by clearing the Bin Counters to 0 to indicate that the bins are empty and all slices of the Agent ID Array are cleared to 1.0. For the first iteration, the top-most slice of the Agent ID Array is bound as the current depth buffer and the Bin Counter is bound as the current color target. The working set, containing all agent IDs, is bound as the input vertex buffer and each element is rasterized as a single point primitive. As each point passes through the vertex shader, the point's screen space position is set by mapping the associated agent's world position to a bin address as mentioned above. The normalized agent ID is stored as the point's depth value. The GPU's depth-test unit is configured to pass fragments that are *less than* the depth value stored in the depth buffer. As a result, of all the agents that map to a given bin, only the agent with the lowest ID (corresponding to the point with the lowest depth value) will be drawn into that bin. Since we can only write a single agent to a given bin per iteration, the pixel shader simply outputs 1 resulting in the bin counter being set to 1 at bins that received an agent. Bins that did not receive any agents will remain set to their initial cleared value of 0. If multiple agents map to a single bin, the agent with the lowest ID will get written and other agents will be rejected to be processed on a subsequent pass.

For the second iteration of the algorithm, the second slice of the Agent ID Array is set as the current depth target and the agents are processed once again. No agents were removed from the working set on the first pass so the second iteration once again takes as input a working set containing all agents. This time the vertex shader does some additional work, it rejects the current point primitive if its agent ID is less than or equal to the ID stored in the previous Agent ID Array slice. Points are marked as "rejected" by setting their depth value to some value outside of the valid depth range. The depth unit is still configured to *less than* function, so, much like depth-peeling [EVERITT01], we are effectively implementing a dual depth buffer which results in the point with the lowest ID that is greater than the previously binned ID to pass. Performing the "greater than" test in the vertex shader rather than the pixel shader allows us to avoid inserting clip/kill instructions in our pixel shader and allows the GPU to perform early-z culling.

After vertex shading, points are passed to a geometry shader. The geometry shader tests the point's depth value and only allows non-rejected points to both be sent to the rasterizer and to be streamed out. Points that are marked for rejection are simply discarded; not rasterized and not streamed out. The pixel shader is set to output 2 so that the Bin Counter will be set to 2 at locations where points are written. At the end of this pass, the resulting stream-out buffer will contain all the agents that were binned during this iteration along with all the agents that have not yet been binned. The stream-out buffer will *not*

contain agents that were binned in the previous iteration since they will have been marked for rejection during the vertex shader's "greater than" test and thus will not have been streamed out in the geometry shader. This stream-out buffer becomes the new working set and is used as input for subsequent iterations of the algorithm.

Subsequent passes follow much like the second iteration of the algorithm. Each time: the depth target is set to the next slice of the agent ID array, the pixel shader is set to output current iteration number, and a new working set is created for use in the next pass. Each iteration results in a reduced working set. The algorithm continues to iterate until the working set is reduced to zero. An overflow condition occurs if the iteration count reaches or exceeds the Agent ID Array depth before the working set is reduced to zero. This can occur if too many agents land in a given bin but in practice overflow can be prevented by using a large enough number of bins so that agents are sufficiently distributed to avoid overflow. Also, the depth of the Agent ID Array can be increased to accommodate higher bin loads.

A ping-ponging technique is used to manage the working set buffers. The "ping" buffer contains the current working set and acts as input during one iteration while the "pong" buffer acts as the output buffer. The roles of the ping and pong buffers are swapped after each iteration.

Two techniques are used to avoid CPU/GPU synchronizations that would result in rendering pipeline stalls. Predicated rendering feature of Direct3D® 10 is used to control the execution of each iteration. Ideally the algorithm should only continue to iterate as long as the previous iteration resulted in a stream-out buffer with non-zero length. Unfortunately if we were to control execution on the CPU by issuing GPU queries after each pass to determine if the algorithm had completed, we would introduce stalls in the rendering pipeline due to CPU/GPU synchronization and this would degrade performance.

To avoid synchronization stalls, all the draw calls for the maximum number of iterations (corresponding to the maximum allowable bin load) are made up front. We still want the algorithm to terminate once all agents have been binned so we use predicated draw calls to terminate upon completion. The draw calls for each iteration are predicated on the condition that the previous iteration resulted in agents being streamed out. If no agents are streamed out then we know the working set has been reduced to zero and we can terminate. Using cascading predicated draw calls in this way will result in the remaining draw calls being skipped. Thus the GPU takes full responsibility for terminating the algorithm once all the agents have been binned. The Direct3D® 10 `DrawAuto` call is used to issue each predicated draw since we do not know the size of the working set from iteration to iteration.

Our spatial data structure provides some benefits over previous techniques [HARADA07]. Querying our data structure is efficient because we store bin loads in a Bin Counter thus allowing us to only read the necessary number of elements from the Agent ID Array and even early-out when a bin is determined to be empty. Additionally our technique provides a mechanism for detecting overflow, employs iterative stream-out reduction of the working set, and gives execution control to the GPU to avoid pipeline stalls.

### 3.2.2.3 *Agent Movement Direction Determination*

Agent's directions need to change as a result of pathfinding and local avoidance models' computations. Each agent evaluates a number of fixed directions relative to the goal direction determined by the global solution. We used five directions in our application. More can be used for increased motion fidelity. The suitable number of directions for our application was determined empirically by evaluating desired motion fidelity versus performance overhead for computing more directions. Each direction is evaluated to determine the time to collision with agents in the current or adjacent bins. Each direction is given a fitness function based on the angle relative to the desired global direction and the time to collision. Time to collision is determined by evaluating a swept circle-circle collision test, in which the radius of each circle is equal to the disc radius *r* of the agents.

The updated velocity (Equation 7) is then calculated based on the direction with the largest fitness function result (Equation 6) and the smallest time to collision in that direction.

$$fitness(\boldsymbol{vp_i}) = w_i t(\boldsymbol{vp}_i) + (\boldsymbol{g}_i \cdot \boldsymbol{vp_i}).5 + .5 \quad (5)$$

$$\boldsymbol{v}_i = \underset{\boldsymbol{vp}_i \in V}{\operatorname{argmax}} fitness(\boldsymbol{vp}_i) \quad (6)$$

$$\boldsymbol{v}_{final} = \hat{\boldsymbol{v}}_i \min(s_a, s_a t(\hat{\boldsymbol{v}}_i)/\nabla ft) \quad (7)$$

where $w_i$ is a per-agent factor affecting the preference to move in the global direction or avoid nearby agents, $t(\boldsymbol{x})$ returns the minimum time to collision with all agents in direction $\boldsymbol{x}$, $V$ is the set of discrete directions to evaluate, $\boldsymbol{g}_i$ is the global navigation direction, $s_a$ is the speed of agent $a$, and $\nabla ft$ is the time-delta since the last simulation frame.

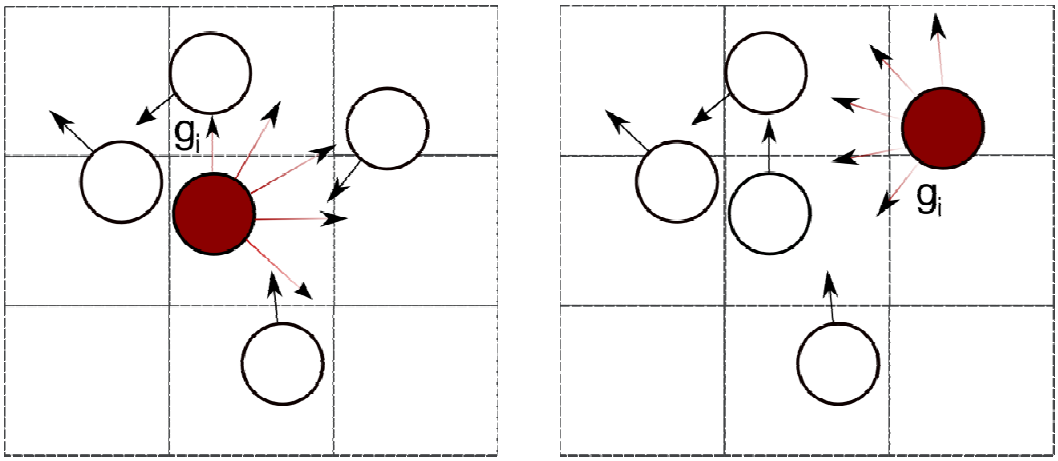


***Figure 6.*** *Each agent evaluates a fixed number of potential movement directions based on the positions and velocities of agents in its current and adjacent bins. Two agents and their corresponding V sets are shown.*

Two example sets of directions to be evaluated for velocity update are shown in Figure 6. Note that the two sets are identical in the local coordinate frame defined by the agent and its global navigation direction $\mathbf{g}_i$. We have also chosen to evaluate only directions that would cause a "right turn" change in

orientation. This eliminates the need to arbitrate which direction an agent should turn based on the velocities of other agents and also results in fewer collision tests. In practice, this limitation is not very noticeable or distracting.

It is important to note that we employ a simple kinodynamic constraint to restrict an agent's change in velocity. It is desired to throttle the change in orientation in a given time step because our agents have a physical limit to how fast they can change their orientations. This prevents sudden broad changes in orientation in dense agent situations.

### 3.2.3 Agent State Management

As agent navigate around the environment, it is important to maintain information about their current state. This includes data such as current position and velocity, current group ID, current animation cycle (or action) and current time within that animation cycle. We also maintain per-agent data such as maximum speed and *goal achievement distance*. Goal achievement distance is a random value is used to determine the distance from the goal at which an agent has reached that goal. This is rather specific to our specific goal types such as mushroom and gold patches where the goal has a specific area and the boundaries are nebulous.

Agent animation cycle transitions are performed using dynamic flow control within the shader controlling agent update logic. If the current time within an animation cycle is greater than the length of the current animation, several conditions are checked to determine what animation cycle should be part of the agent's state. These are: current animation cycle, distance from nearest goal, number of agents nearby, distance from fear inducing obstacles such as the ghost froblin and noxious gas clouds, and current group. While these agent updates will not be very coherent, the agent data texture is very small and the performance impact is slight. Despite the dimensionality of input to the animation transition function, it may be possible to precompute the animation transition logic into textures (as done in [MHR07]) and eliminate flow control.

### 3.2.4 *Pathfinding Results Discussion*

Our approach has been used to simulate ~65,000 agents at interactive rates on an ATI Radeon™ HD 4870 while performing intensive rendering tasks such as multi-million triangle scene rendering, global illumination approximation, atmospheric scattering, and high-quality cascaded shadow mapping. All results were collected on an AMD Phenom™ X4 Quad-Core CPU system with 2GB of RAM and an ATI Radeon™ 4870 graphics card with 512MB of GDDR5 video memory and standard engine and memory clocks. The main bottleneck in our application is rendering a massive number of agents. In the case of simulating 65K agents, we use a simplified agent model (a cylinder). Simulation (global and local) alone for 65K agents on the above system is 45 fps. Simulation of crowd behavior and interaction along with rendering for this large number of agents is 31 fps. Note that even with using a very simple cylinder model, due to extremely large number of agents, we are rendering 9.8M triangles in the latter case. All of our testing results were collected rendered at HD resolution with 4X MSAA.

While we chose to use the two crowd behavior simulation techniques for our global and local navigation, they also can be used separately as they each have their own advantages and disadvantages.

The continuum approach used for our global solver works well in the *Froblins* scenario where there are large numbers of agents with only a few types of goals. More complex scenarios with diverse tasks are likely to be incompatible with this type of approach. Another disadvantage of using a continuum approach is that all agents are modeled as having global knowledge. An agent will make navigation choices based on obstacles that are not visible to it, which may also not be desired. We feel that the continuum approach would be excellent for *ambient crowds*. That is, large groups of non-player characters which are present mostly for scenery but are expected to navigate around a dynamic environment or moving characters.

The local model also has disadvantages. By limiting local avoidance velocities to be of a clockwise nature, the variation in agent interaction is somewhat limited. Using a small discrete set of local directions for navigation can also lead to oscillation between two directions, creating distracting behavior. While the local avoidance model will prevent collisions in very densely packed situations, scenarios arise where agents can deadlock and will become stuck. This typically happens at sinks in agent navigations such as at a small goal. Once agents become densely packed around a goal, agents that reach the goal will be unable to navigate out of the goal area. This could be solved by incorporating varying levels of aggressive behavior into agent movement that causes agents to push each other out of the way. This type of approach could be augmented by a composite agent approach [YCP*08] to “trail-blaze” paths through densely packed agents.

## 3.3 Character LOD Management

The overarching goal of our system is simulation and rendering of massive crowds of characters with high level of detail. The latest generations of commodity GPUs demonstrate incredible increases in geometry performance, especially with the inclusion of GPU tessellation pipeline (Section 3.5). Nevertheless, even with state-of-the-art graphics hardware, rendering multiple thousands of complex characters with high polygonal counts at interactive rates is very taxing. Rendering thousands of characters with over a million of polygons each is neither practical, nor wise, as in many cases these characters may be very small on the screen and therefore performance is wasted on the details that go unnoticed. For this reason it is essential to use culling and level of detail (LOD) techniques in order to make this rendering problem tractable.

Culling and LOD management have traditionally been CPU-centric tasks, trading a modest amount of CPU overhead for a much larger reduction in the GPU workload. However, a common difficulty arises when the positional data is generated by a GPU-based simulation, and, therefore, would require a costly read-back operation for CPU-side scene management. This is exactly the situation we’ve encountered, as we simulated and animated our characters entirely on the GPU.

The alternative is to use the available compute to perform all culling and scene management directly on the GPU. In our *Froblins* demo, we solve this problem by employing Direct3D® 10 geometry shaders in a

novel way to perform character culling and LOD sorting entirely on the GPU. This enables us to perform these tasks efficiently for GPU-simulated characters. The underlying ideas could also be applied with a CPU simulation, in order to offload the scene management from the CPU.

### 3.3.1 Using Stream-Out Operations as *Filtering*

Our system takes advantage of *instancing* support available with Direct3D® 10 and Direct3D® 10.1 API. We render an army of characters as varied instanced characters, with individual actions and animations controlled on the GPU. This naturally leads to the key idea behind our scene management approach: the use of geometry shaders that act as *filters* for a set of character instances. A filtering shader works by taking a set of point primitives as input, where each point contains the per-instance data needed to render a given character (position, orientation, and animation state). The filtering shader re-emits only those points which pass a particular test, while discarding the rest. The emitted points are streamed into a buffer which can then be re-bound as instance data and used to render the characters. Multiple filtering passes can be chained together by using successive *DrawAuto* calls, and different tests can be set up simply by using different shaders.

In practice, we use a shared geometry shader to perform the actual filtering, and perform the different filtering tests in vertex shaders. Aside from providing more modular code, this approach can also provide performance benefits. The source to this filtering geometry shader is shown in Listing 1.

```
struct GSInput
{
    // X,Y,Z contain the character's origin in world space
    // W contains a group number, which is used to vary character appearance
    float4 vPositionAndGroup : PositionAndGroup;

    // X,Y contain the character orientation (a vector in the X/Z plane)
    // Z contains the index of the character's animation cycle
    // W contains the time along the cycle (see section 3.5)
    float4 vDirection        : DirectionStateAndTime;

    // Result of predicate test:  1 == emit, 0 == do not emit
    float  fResult           : TestResult;
};

struct GSOutput
{
    float4 vPositionAndGroup : PositionAndGroup;
    float4 vDirection        : DirectionStateAndTime;
};


[maxvertexcount(1)]
void main( point GSInput vert[1], inout PointStream<GSOutput> outputStream )
{
    [branch]
    if( vert[0].fResult == 1 )
    {
        GSOutput o;
        o.vPositionAndGroup = vert[0].vPositionAndGroup;
        o.vDirection        = vert[0].vDirection;
        outputStream.Append( o );
    }
}
```

***Listing 1.*** *A stream-filtering geometry shader*

### 3.3.2 View-Frustum Culling

It is straightforward to perform view frustum culling using a filtering geometry shader, as described above. For view-frustum culling, the vertex shader simply performs an intersection check between the character bounding volume and the view frustum, using the usual algorithms (for example, [AHH08]). If the test passes, then the corresponding character is visible, and its instance data is emitted from the geometry shader and streamed out. Otherwise, it is discarded. An example of a culling vertex shader is given in Listing 2.

```
struct VSInput
{
   float4 vPositionAndGroup : PositionAndGroup;
   float4 vDirectionStateAndTime : DirectionStateAndTime
};

struct VSOutput
{
    float4 vPositionAndGroup : PositionAndGroup;
    float4 vDirection        : DirectionStateAndTime;
    float  fVisible          : TestResult;
};

// Computes signed distance between a point and a plane
// vPlane: Contains plane coefficients (a,b,c,d) where: ax + by + cz = d
// vPoint: Point to be tested against the plane.
float DistanceToPlane( float4 vPlane, float3 vPoint )
{
    return dot( float4( vPoint, -1 ), vPlane );
}

// Frustum cullling on a sphere.  Returns 1 if visible, 0 otherwise
float CullSphere( float4 vPlanes[6], float3 vCenter, float fRadius )
{
    for( uint i=0; i<6; i++ )
    {
        // entire sphere is outside one of the six planes, cull immediately
        if( DistanceToPlane( vPlanes[i], vCenter ) > fRadius )
            return 0;
    }
    return 1;
}

float4 vFrustumPlanes[6];  // view-frustum planes in world space (normals face out)
float3 vSphereCenter;      // bounding sphere center, relative to character origin
float  fSphereRadius;      // bounding sphere radius

VSOutput VS( VSInput i )
{
    // compute bounding sphere center in world space
    float3 vObjectPosWS    = i.vPositionAndGroup.xyz;
    float3 vSphereCenterWS = vBoundingSphereCenter.xyz + vObjectPosWS;

    // perform view-frustum test
    float fVisible = CullSphere( vFrustumPlanes, vSphereCenterWS, fSphereRadius );

    VSOutput o;
    o.vPositionAndGroup      = i.vPositionAndGroup;
    o.vDirectionStateAndTime = i.vDirectionStateAndTime;
    o.fVisible               = fVisible;
    return o;
}
```

***Listing 2.*** *Vertex shader for view-frustum culling*

### 3.3.3 Occlusion Culling

We can also perform occlusion culling in this framework, to avoid rendering characters which are completely occluded by mountains or structures. Because we are performing our character management on the GPU, we are able to perform occlusion culling in a novel way, by taking advantage of the depth information that exists in the hardware Z buffer. This approach requires far less CPU overhead than an approach based on predicated rendering or occlusion queries, while still allowing culling against arbitrary, dynamic occluders. Our approach is similar in spirit to the hierarchical Z testing that is implemented in modern GPUs, and was inspired by the work of [GKM93], who used a hierarchical depth image combined with an octree to cull occluded geometry in bulk.

After rendering all of the occluders in the scene, we construct a hierarchical depth image from the Z buffer, which we will refer to as a *Hi-Z map*. The Hi-Z map is a mip-mapped, screen-resolution image, where each texel in mip level *i* contains the maximum depth of all corresponding texels in mip level *i-1*. In the most detailed mip level, each texel simply contains the corresponding depth value from the Z buffer. This depth information can be collected during the main rendering pass for the occluding objects; a separate depth pass is not required to build the Hi-Z map.

After construction of the Hi-Z map, occlusion culling can be performed by examining the depth information for pixels which are covered by an object's bounding sphere, and comparing the maximum fetched depth to the projected depth of a point on the sphere that is nearest to the camera. Although this approach does not provide an exact occlusion test, it gives a conservative estimate that works well in many cases, and will never result in false culling.

#### 3.3.3.1 *Hi-Z Map Construction*

For single-sample rendering, one can use the Hi-Z map as the main depth buffer for rendering the scene (using a DepthStencil view of the first mip level). In Direct3D® 10.1, multi-sampled depth buffers can also be supported, with an extra full-screen quad pass, by first computing the maximum depth of each pixel's sub-samples and storing the result in the lowest level of the Hi-Z map.

Subsequent levels are generated using a sequence of reduction passes, which repeatedly fetch texels and compute their maximum, as shown in Listing 3. Because screen-sized images typically do not mip well, care must be taken when reducing odd-sized mip levels. In this case, the pixels on the odd-sized boundary edge must fetch additional texels to ensure that their depth values are taken into account. In addition, it is necessary to use integer calculations for the texture address arithmetic, because floating-point error can result in incorrect addressing when rendering into the lower mip levels.

Each of the reduction passes renders into one mip level of the Hi-Z map resource, while sampling from the previous one. This is valid approach in Direct3D® 10, as long as the resource view used for the input mip level does not overlap the one being used for output (different input and output views must be created for each pass).

```
struct PSInput
{
    // Fractional pixel coordinates (0.5, 1.5, 2.5, etc…)
    float4 vPositionSS : SV_POSITION;

    // Dimensions of 'tCurrentMip'.
    //  Can be obtained by calling 'GetDimensions' in the vertex shader.
    nointerpolation uint2 vLastMipSize : DIMENSION;
};

Texture2D<float> tCurrentMip;
sampler          sPoint;

float4 main( PSInput i ) : SV_TARGET
{
    // get integer pixel coordinates
    uint3 nCoords      = uint3( i.vPositionSS.xy, 0 );
    uint2 vLastMipSize = i.vLastMipSize;

    // fetch a 2x2 neighborhood and compute the max
    nCoords.xy *= 2;

    float4 vTexels;
    vTexels.x = tCurrentMip.Load( nCoords );
    vTexels.y = tCurrentMip.Load( nCoords, uint2(1,0) );
    vTexels.z = tCurrentMip.Load( nCoords, uint2(0,1) );
    vTexels.w = tCurrentMip.Load( nCoords, uint2(1,1) );

    float fM = max( max( vTexels.x, vTexels.y ), max( vTexels.z,vTexels.w ) );

    // if we are reducing an odd-sized texture,
    // then the edge pixels need to fetch additional texels
    float2 vExtra;
    if( (vLastMipSize.x & 1) && nCoords.x == vLastMipSize.x-3 )
    {
        vExtra.x = tCurrentMip.Load( nCoords, uint2(2,0) );
        vExtra.y = tCurrentMip.Load( nCoords, uint2(2,1) );
        fM = max( fM, max( vExtra.x, vExtra.y ) );
    }

    if( (vLastMipSize.y & 1) && nCoords.y == vLastMipSize.y-3 )
    {
        vExtra.x = tCurrentMip.Load( nCoords, uint2(0,2) );
        vExtra.y = tCurrentMip.Load( nCoords, uint2(1,2) );
        fM = max( fM, max( vExtra.x, vExtra.y ) );
    }

    // extreme case:  If both edges are odd, fetch the bottom-right corner texel
    if( ( ( vLastMipSize.x & 1 ) && ( vLastMipSize.y & 1 ) ) &&
        nCoords.x == vLastMipSize.x-3 && nCoords.y == vLastMipSize.y-3 )
    {
        fM = max( fM, tCurrentMip.Load( nCoords, uint2(2,2) ) );
    }

    return fM;
}
```

***Listing 3.*** *Pixel shader used for Hi-Z map construction*

### 3.3.3.2 *Culling with the Hi-Z Map*

Once we have constructed the Hi-Z map, we perform another stream filtering pass which uses this information to perform occlusion culling. In order to ensure a stable frame rate, it is desirable to restrict the number of fetches that are performed for each character, and to avoid divergent flow control

between character instances. We can accomplish this goal by exploiting the hierarchical structure of the Hi-Z map.

We first compute the bounding square in image space which fully encloses the character's projected bounding sphere. We then select a specific mip level in the Hi-Z map at which the square will cover no more than one 2×2 texel neighborhood. This 2×2 neighborhood is then fetched from the map, and the depth values are compared against the projected depth of a point on the bounding sphere that is nearest to the camera. The structure of the Hi-Z map guarantees that if any of these texels occludes the object, then all texels beneath it will also occlude. Although we have chosen to use a 2×2 neighborhood, a larger one could be used instead, and would provide more effective culling at the expense of added overhead in the culling test.

To obtain the closest point on the bounding sphere, one can simply use the following formula:

$$Pv = Cv - \left(\frac{Cv}{|Cv|}\right) r \qquad (8)$$

Here, $P_v$ is the closest point in camera space, $C_v$ is the sphere center in camera space, and $r$ is the sphere radius. The projected depth of this point will be used for the depth comparisons ahead. Note that if the camera is inside the bounding sphere, this formula will result in a point behind the near plane, whose projected depth is not well defined. In this case, we must refrain from culling the character to prevent a false occlusion.

To compute the character's bounding square, we first calculate its projected height in screen space based on its distance from the image plane. Note that we define screen space as the space obtained after perspective projection. The height in screen space is given by:

$$h = \frac{r}{d \tan\left(\frac{\theta}{2}\right)} \qquad (9)$$

where $d$ is the distance from the sphere center to the image plane and $\Theta$ is the vertical field of view of the camera. The width of the bounding square is equal to this height divided by the aspect ratio of the back buffer. The size of the square in screen space is equal to twice its size in NDC space (which is a normalized space starting at the top-left corner of the screen). Note also that, for non-square resolutions, a square on the screen is actually a rectangle in screen space and NDC space.

When sampling the Hi-Z map, we would like to fetch from the lowest level in which the bounding square covers at most four texels. This will allow us to use a fixed number of fetches to reject any square, no matter its size. To choose the level, we need only ensure that the size of the square is smaller than the size of a single texel at the chosen resolution. In other words, we choose the lowest level *i* such that:

$$\left(\frac{W}{2^i}\right) < 1 \qquad (10)$$

where the width of the rectangle in pixels is $W$. This yields the following equation for *i*:

$$i = \lceil \log_2(W) \rceil \qquad (11)$$

This holds, provided that the width of the rectangle in pixels is larger than the height. If this is not the case, the height in pixels should be used instead. This will happen whenever the aspect ratio is less than one.

Once we have chosen the correct mip level, we perform a texture fetch from the Hi-Z map at each corner of the bounding square, compute the maximum fetched value, and compare it with the depth of the depth of the point $P_v$.

HLSL code for occlusion culling is given in Listing 4. The vertex shader shown in the listing is used together with a filtering geometry shader (Listing 1) to filter out character instances which are occluded by other scene elements. Remember that this calculation is only performed once per character, not once per rendered vertex.

### 3.3.4 LOD Selection

Given the above, LOD selection is also simple to implement. We use a discrete LOD scheme, in which a different level of detail is selected based on the distance from the camera to the character's center. This is implemented by using three successive filtering passes to separate the characters into three disjoint sets, based on their distances to the camera. These filtering passes are applied to the results of the culling steps, so that only visible characters are processed. The culling results are computed once, and re-used for the LOD selection passes. We render the characters in the finest (closest) LOD using hardware tessellation and displacement mapping (see section 3.5), and use conventional rendering for the middle LOD, and simplified geometry and pixel shaders for the furthest LOD.

```
float4x4 mV;              // Viewing transform
float4x4 mP;              // Projection transform
float3   vCameraPosition; // Camera location in world space
float    fCameraFOV;      // Camera's vertical field of view angle
float    fCameraAspect;   // Camera aspect ratio
float4   vSphere;         // Bounding sphere center (XYZ) and radius (W), object space
float4   vViewport;       // X,Y,Width,Height
Texture2D<float> tHiZMap;
sampler          sHiZPoint;

struct VSInput
{
   float4 vPositionAndGroup   : PositionAndGroup;
   float4 vDirection          : DirectionStateAndTime;
};
struct VSOutput
{
   float4 vPositionAndGroup : PositionAndGroup;
   float4 vDirection        : DirectionStateAndTime;
   float  fVisible          : IsVisible;
};
VSOutput VS( VSInput i )
{
   VSOutput o;
   o.vPositionAndGroup = i.vPositionAndGroup;
   o.vDirection.xyzw   = i.vDirection.xyzw;
   o.fVisible          = 1;

   // compute bounding sphere center in camera space
   float3 vAgentCenterWS = vSphere.xyz + i.vPositionAndGroup.xyz;
   float3 Cv = mul( mV, float4( vAgentCenterWS.xyz, 1 ) ).xyz;

   // Do not cull agents if the camera is inside their bounding sphere
   if( length( Cv ) > vSphere.w )
   {
      // compute nearest point to camera on sphere, and project it
      float3 Pv          = Cv - normalize( Cv ) * vSphere.w;
      float4 vPositionSS = mul( mP, float4(Pv,1) );

      // compute radii of bounding rectangle in screen space (2x the radii in NDC)
      float fRadiusY = vSphere.w / ( Cv.z * tan( fCameraFOV / 2 ) );
      float fRadiusX = fRadiusY / fCameraAspect;

      // compute UVs for corners of projected bounding square
      float2 vCornerNDC  = vPositionSS.xy / vPositionSS.w;
             vCornerNDC  = float2(0.5,-0.5)* vCornerNDC + float2( 0.5, 0.5 );
             vCornerNDC -= 0.5 * float2( fRadiusX, fRadiusY );

      float2 vCorner0 = vCornerNDC;
      float2 vCorner1 = vCornerNDC + float2( fRadiusX, 0 );
      float2 vCorner2 = vCornerNDC + float2( 0,        fRadiusY );
      float2 vCorner3 = vCornerNDC + float2( fRadiusX, fRadiusY );

      // Choose a MIP level in the HiZ map (assume that width > height)
      float W    = fRadiusX * vViewport.z;
      float fLOD = ceil( log2( W ) ) ;

      // fetch depth samples at the corners of the square to compare against
      float4 vSamples;
      vSamples.x = tHiZMap.SampleLevel( sHiZPoint, vCorner0, fLOD );
      vSamples.y = tHiZMap.SampleLevel( sHiZPoint, vCorner1, fLOD );
      vSamples.z = tHiZMap.SampleLevel( sHiZPoint, vCorner2, fLOD );
      vSamples.w = tHiZMap.SampleLevel( sHiZPoint, vCorner3, fLOD );
      float fMaxDepth = max( max( vSamples.x, vSamples.y ),
                             max( vSamples.z, vSamples.w ) );
      // cull agent if the agent depth is greater than the largest of our ZMap values
      o.fVisible = ( (vPositionSS.z / vPositionSS.w) > fMaxDepth  ) ? 0 : 1;
   }
   return o;
}
```

***Listing 4.*** *Vertex shader for per-instance occlusion culling*

### 3.3.5 Character Management System High Level Overview

Using the above concepts, we can now describe our GPU character management system in its entirety, illustrated in pseudo-code in Listing 5. We begin by rendering the occluding geometry and preparing the Hi-Z map. We then run all characters through the view frustum culling filter, and stream out the ones which pass. The results of the view-frustum pass are then run through the occlusion culling filter using a *DrawAuto* call. The instances which pass the occlusion culling test are then run through a series of LOD selection filters to separate them by LOD.

Once we've determined the visible characters in each LOD, we would like to render all of the character instances in each given LOD. In order to issue the draw call for that LOD, we need to know the instance count. Obtaining this instance count unfortunately requires the use of a stream out statistics query. Like occlusion queries, stream out statistics queries can cause significant stalls, and, thus, performance degradation, when the results are used in the same frame that the query is issued, because the GPU may go idle while the application is processing the query results. However, an easy solution for this is to re-order draw-calls to fill the gap between previous computations and the result of the query. In our system, we are able to offset the GPU stall by interleaving scene management with the next frame's crowd movement simulation. This ensures that the GPU is kept busy while the CPU is reading the query result and issuing the character rendering commands.

```
RenderOccluders()
RenderHiZMap()

// Do view-frustum culling
// streaming visible instances to 'frustumCullOutput' buffer
BindFrustumShader()
IASetVertexBuffers( characterVB );
SOSetTargets( frustumCullOutput );
Draw( POINT_LIST, CHARACTER_COUNT );

// Do occlusion culling on frustum culling results
//  streaming visible instances to 'occlusionCullOutput' buffer
BindOcclusionShader();
IASetVertexBuffers( frustumCullOutput );
SOSetTargets( occulsionCullOutput );
DrawAuto( POINT LIST ) // render output of frustum culling shader

// Filter occlusion culling results by LOD, and issue queries to read the final counts
IASetVertexBuffers( occlusionCullOutput );
for(int i=0; i<LOD_COUNT; i++ )
{
    BindLODShader( LOD[i].minDistance, LOD[i].maxDistance );
    SOSetTargets( LOD[i].instanceDataBuffer );

    LOD[i].query->Begin()
    DrawAuto( POINT_LIST ); // render output of occlusion culling shader
    LOD[i].query->End()
}

// if possible, do other CPU and GPU work here, to fill out the query stall

// read back character counts and render characters in each LOD
for( int i=0; i<LOD_COUNT; i++ )
{
    int instanceCount = LOD[i].query->GetPrimitiveCount()
    DrawInstancedCharacter( LOD[i], instanceCount );
}
```

***Listing 5.*** *Summary of our character management system*

## 3.4 Character Animation

The traditional approach to rendering key framed, skinned characters is to sample the animations and compute a matrix palette on the CPU, which is then loaded into constant store for consumption by vertex shaders. This is generally done once per character. Although it is sometimes possible to pack the bones for multiple individuals into constant store, there are still serious limitations on the number of characters that can be handled using this approach, and large crowds of characters will still require numerous draw calls. Furthermore, since we are using the GPU to manage our characters (see sections 3.2 and 3.3), the traditional approach to skinning is simply not feasible in our case. We solve this problem in our system by moving the animation sampling onto the GPU.

Our agents can perform a set of predefined actions (walking, eating, mining, etc.,), some of which are demonstrated in Figure 7. Each action has an associated animation sequence. In our system we use close to 40 different animation sequences and transitions. During animation preprocessing, we flatten the transformation hierarchy and compute a bone transformation for each key frame that transforms that bone directly into object space. During the simulation, each character is assigned an animation sequence, and a time offset within that sequence. During character rendering, the vertex shader uses this information to fetch, interpolate, and blend the key frames for each bone. Each instanced character performs its skinning in object space, and then transforms the result according to its position and orientation.

***Figure 7.*** *Example of different actions that our Froblins can perform. These actions are controlled by the character logic shader, which also determines the current animation sequence based on each character's current state and desired action. From the left: (a) Froblin carrying his hard-earned treasure to the drop-off location; (b) User placed a noxious poison cloud in the path of Froblins and as a result they scatter away. Here we see the critter running away from the hazard; (c) The Froblin is about to munch on some delicious mushrooms; (d) A bit of peaceful resting restores this Froblin's good spirits.*

The layout of our animation data is illustrated in Figure 8. The transformations are stored as 3×4 matrices. We use a texture array, where the horizontal and vertical dimensions correspond to key frame and bone index, and the slice number is used to index the animation sequence. Varying the time along one axis of the texture allows us to use the texture filtering hardware to interpolate between the key frames. Shader code to perform the animation fetch and blending is given in Listing 6. Note that we sort our per-vertex bone influences by weight, and use dynamic branching to avoid fetching zero-weight bones. In our case, this provides a notable performance gain, as most of our vertices do not possess more than two bone influences.

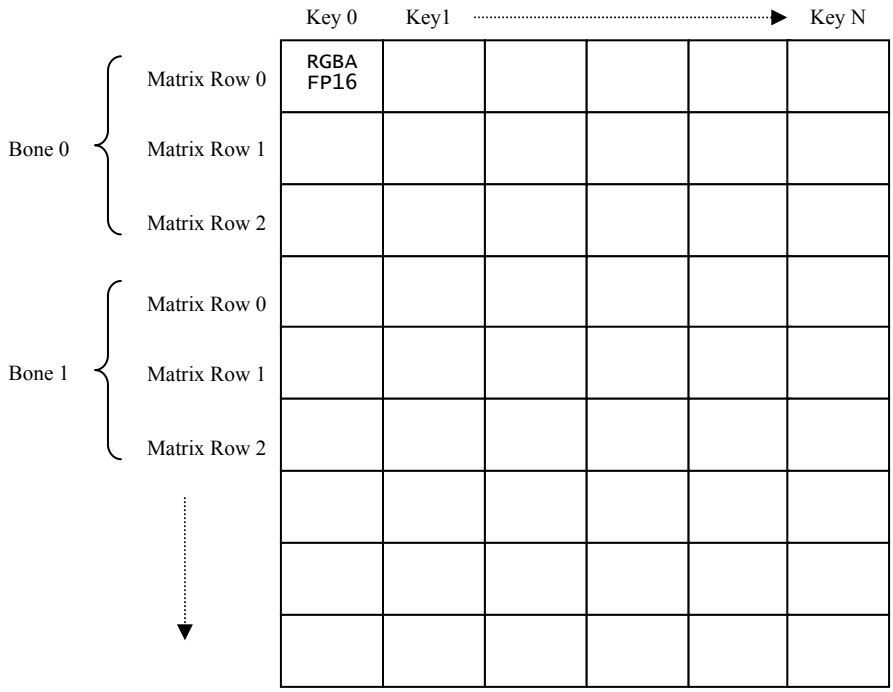


***Figure 8.*** *Animation texture layout*

In our case, the amount of memory consumed by a full set of character animation data is about 8 MB, which is a reasonable size for our particular application. Unfortunately, a large fraction of this is wasted space, which is incurred because the width of the texture must be large enough to accommodate the longest animation sequence. For future directions, we would like to investigate the use of GPU-friendly sparse textures to store the animation data. In our case, we find that most of the animations are fairly short, with only a few long outliers. This waste could be significantly reduced by simply packing multiple short animation sequences into one page of the texture array, and adding a lookup table to the shader which stores the start location for each sequence. Another, simpler solution might be to continue using one sequence per page, but to separate short and long sequences into separate arrays. We did not pursue either of these solutions in our implementation because they would have introduced additional complexity to the shaders, and memory consumption was not enough of an issue to justify the possible performance loss.

```
float fTexWidth;
float fTexHeight;
float fCycleLengths[MAX_SLICE_COUNT];

Texture2DArray<float4> tBones;
sampler                sBones; // should use CLAMP addressing and linear filtering
```

```
void SampleBone( uint nIndex, float fU, uint nSlice,
                 out float4 vRow1, out float4 vRow2, out float4 vRow3 )
{
    // compute vertical texture coordinate based on bone index
    float fV = (nIndices[0]) * (3.0f / fTexHeight);

    // compute offsets to texel centers in each row
    float fV0 = fV + ( 0.5f / fTexHeight );
    float fV1 = fV + ( 1.5f / fTexHeight );
    float fV2 = fV + ( 2.5f / fTexHeight );

    // fetch an interpolated value for each matrix row, and scale by bone weight
    vRow1 = fWeight * tBones.SampleLevel( sBones, float3( fU, fV0, nSlice ), 0 );
    vRow2 = fWeight * tBones.SampleLevel( sBones, float3( fU, fV1, nSlice ), 0 );
    vRow3 = fWeight * tBones.SampleLevel( sBones, float3( fU, fV1, nSlice ), 0 );
}

float3x4 GetSkinningMatrix( float4 vWeights, uint4 nIndices, float fTime, uint nSlice )
{
    // derive length of longest packed animation
    float fKeyCount       = fTexWidth;
    float fMaxCycleLength = fKeyCount / SAMPLE FREQUENCY;

    // compute normalized time value within this cycle
    // if out of range, this will automatically wrap
    float fCycleLength = fCycleLengths[ nSlice ];
    float fU = frac( fTime / fCycleLength );

    // convert normalized time for this cycle into a texture coordinate for sampling.
    //  We need to scale by the ratio of this cycle's length to the longest,
    //   because the texture size is defined by the length of the longest cycle
    fU *= (fCycleLength / fMaxCycleLength);

    float4 vSum1, vSum2, vSum3;
    float4 vRow1, vRow2, vRow3;

    // first bone
    SampleBone( nIndices[0], fU, nSlice, vSum1, vSum2, vSum3 );
    vSum1 *= vWeights[0];
    vSum2 *= vWeights[0];
    vSum3 *= vWeights[0];

    // second bone
    SampleBone( nIndices[1], fU, nSlice, vRow1, vRow2, vRow3 );
    vSum1 += vWeights[1] * vRow1;
    vSum2 += vWeights[1] * vRow2;
    vSum3 += vWeights[1] * vRow3;

    // third bone
    if( vWeights[2] != 0 )
    {
        SampleBone( nIndices[2], fU, nSlice, vRow1, vRow2, vRow3 );
        vSum1 += vWeights[2] * vRow1;
        vSum2 += vWeights[2] * vRow2;
        vSum3 += vWeights[2] * vRow3;
    }

    // fourth bone
    if( vWeights[3] != 0 )
    {
        SampleBone( nIndices[3], fU, nSlice, vRow1, vRow2, vRow3 );
        vSum1 += vWeights[3] * vRow1;
        vSum2 += vWeights[3] * vRow2;
        vSum3 += vWeights[3] * vRow3;
    }

    return float3x4( vSum1, vSum2, vSum3);
}
```

***Listing 6.*** *Shader code to fetch, interpolate, and blend bone animations*

## 3.5 Tessellation and Crowd Rendering

***Figure 9.*** *Our system allows rendering characters with extreme details in close-up when using tessellation (left). On the right, the same character is rendered without the use of tessellation using identical pixel shaders and textures. While using the same memory footprint, we are able to add high level of details for the tessellated character on the left, whereas the low resolution character has much coarser silhouettes.*

Recent generations of GPU architecture such as Xbox® 360 and ATI Radeon® HD 2000, 3000 and 4000 series have shown tremendous improvements in geometry processing. These include unified shader architecture (introduced with Xbox® 360), increased number of dedicated shader units, and hardware tessellation pipeline. Furthermore, with the introduction of upcoming graphics APIs such as Direct3D® 11 (as described in [KLEIN08] and [GEE08]), tessellation and displacement mapping will be universally supported across all hardware platforms designed for that generation and, thus, solidify tessellation as the first-class citizen feature in the real-time domain. Next generation games, including those authored for Xbox® 360 will use tessellation for extreme visual impact, high quality, and stable performance. A strong understanding of how this technology works is the key to quick and successful adoption.

| | Polygons | Total Memory |
|---|---|---|
| **Froblin control cage, Low resolution model** | 5,160 faces | Vertex and index buffers: 100KB<br><br>2K × 2K 16 bit displacement map: 10MB |
| **Zbrush® high resolution Froblin model** | 15+ M faces | Vertex buffer: ~270 MB<br><br>Index Buffer: 180 MB |

***Table 1.*** *Comparison of memory footprint for high and low resolution models for our main character.*

***Figure 10.*** *Comparison of low resolution model (top) and high resolution model (bottom) for the Froblin character.*

Hardware tessellation provides several key benefits specifically crucial for interactive systems, such as video games. One of the main advantages is effective compression of vertex data. When using tessellation, we are specifying the surface topology, parameterization and animation data for the coarse control mesh. This mesh is authored to have low amount of detail, and just to capture the overall shape of the desired object. We can then combine rendering of this control cage with GPU tessellation and displacement mapping to greatly increase the overall amount of details. High frequency details such as wrinkles, bumps and dents are captured by the displacement map. Figure 10 shows an example of this for our main character. Thus using tessellation allows us to reduce memory footprint and bandwidth. This is true both for on-disk storage and for system and video memory footprint, thus reducing the overall game distribution size, improving loading time. The memory savings are especially relevant for console developers, where memory resources are scarce. Table 1 demonstrates memory savings for our main character, the Froblin. Additional overview of the benefits provided by tessellation can be found in [TATARCHUK08].

### 3.5.1 GPU Tessellation Pipeline

In this section we will provide an overview of GPU tessellation pipeline available on current consumer hardware, as used in our system. We designed an API for a GPU tessellation pipeline taking advantage of hardware fixed-function tessellator unit available on recent consumer GPUs. The tessellation process is outlined in Figure 11.

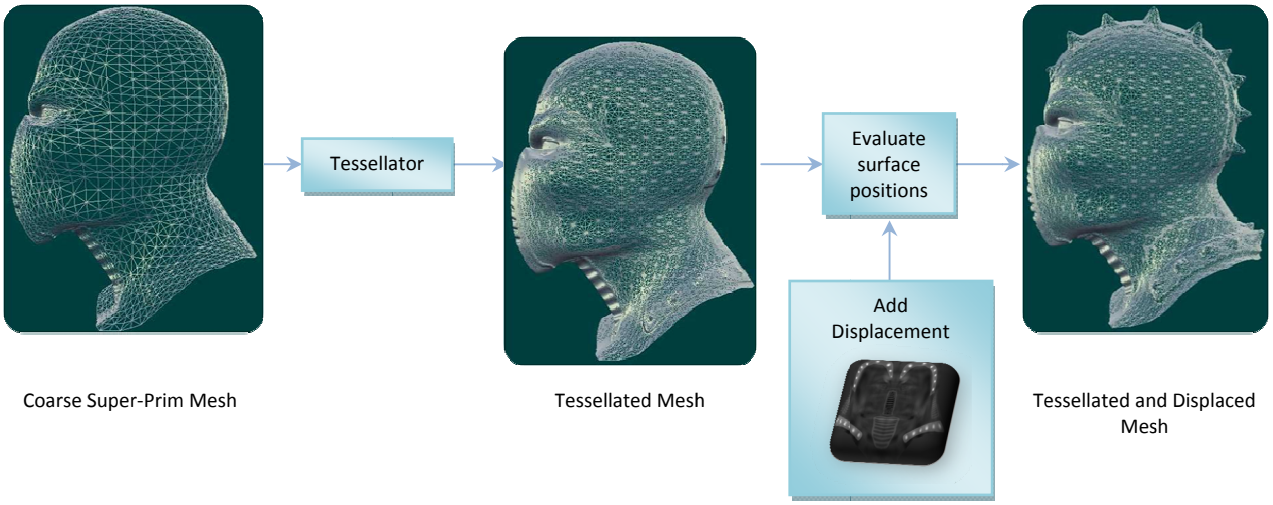


***Figure 11.*** *An overview of the tessellation process. We start by rendering a coarse, low resolution mesh (also referred to as the "control cage" or "the super-primitive mesh"). The tessellator unit generates new vertices, thus* amplifying *the input mesh. The vertex shader is used to evaluate surface positions and add displacement, obtaining the final tessellated and displaced high resolution mesh seen on the right.*

Going through the process for a single input polygon, we have the following: the hardware tessellator unit takes an input primitive (which we refer to as a *super-primitive*), and amplifies it (up to 411 triangles, or 15X times for Xbox® 360 or ATI Radeon® HD 2000-4000 GPU generations, or 64X for Direct3D® 11 generation of graphics cards). A vertex shader (which we refer to as an *evaluation shader*) is invoked for

each tessellated vertex and is provided with the vertex indices of the super-primitive, and the barycentric coordinates of the vertex. The evaluation shader uses this information to calculate the position of the tessellated vertex, using whatever technique it wishes. The level of tessellation can be controlled either by a per-draw call tessellation factor, or by providing per-edge tessellation factors in a vertex buffer for each triangle edge in the input mesh. We recommend the interested reader look to [TATARCHUK08] for more details about the specific capabilities of the GPU tessellation on current and future hardware.

### 3.5.2 Rendering Characters with Interpolative Tessellation

We use interpolative planar subdivision with displacement to efficiently render our highly detailed characters. We specify tessellation level, controlling the amount of amplification, per draw-call. Therefore, we can use tessellation to control how fine we are going to subdivide this character's mesh. We can use the information about character location on the screen or other factors to control the desired amount of details. Furthermore, we use the same art assets for rendering the tessellated character as for the regular, conventional rendering used in current games.

Combining tessellation with instancing allows us to render diverse crowds of characters with minimal memory footprint and bandwidth utilization. By storing only a low-resolution model (5.2K triangles), and applying a displacement map in the evaluation shader, we can render a detail-rich, 1.6M triangle character using very little memory. We can also limit per-vertex animation computations to the original mesh, since we only need to store animation data for the control cage of the character. GPU tessellation allows us to provide the data to GPU at coarse resolution, while rendering with high levels of detail. Listing 7 provides an example of the vertex shader we used for evaluating the resulting surface positions.

```
struct VSInput
{
   float3 vPositionOS_vert0      : POSITION0;
   float3 vNormalOS_vert0        : NORMAL0;
   float3 vTangentOS_vert0       : TANGENT0;
   float3 vBinormalOS_vert0      : BINORMAL0;
   float2 vUV_vert0              : TEXCOORD0;

   float3 vPositionOS_vert1      : POSITION1;
   float3 vNormalOS_vert1        : NORMAL1;
   float3 vTangentOS_vert1       : TANGENT1;
   float3 vBinormalOS_vert1      : BINORMAL1;
   float2 vUV_vert1              : TEXCOORD1;

   float3 vPositionOS_vert2      : POSITION2;
   float3 vNormalOS_vert2        : NORMAL2;
   float3 vTangentOS_vert2       : TANGENT2;
   float3 vBinormalOS_vert2      : BINORMAL2;
   float2 vUV_vert2              : TEXCOORD2;

   float3 vInstancePosWS : WSInstancePosition; // This is per-instance data

   float3 vBarycentric: TessCoordinates;// Tessellation-specific system-generated values
};
struct VSOutput
{
   float3 vNormalWS     : Normal;
   float3 vTangentWS    : Tangent;
   float3 vBinormalWS   : Binormal;
   float2 vUV           : TEXCOORD0;
   float fGroup         : GroupID;
   float4 vPositionWS   : Position;
   float4 vPositionSS   : SV POSITION;
};

float4x4          mVP;
Texture2D<float>  tDisplacement;
SamplerState      sPointClamp;
SamplerState      sBaseLinear;
float             fDisplacementScale;
float             fDisplacementBias;

VSOutput VS( VSInput i )
{
   VSOutput o;

   //Interpolated tessellated vertex:..............................................
   float3 vPositionOS = i.vPositionOS_vert0 * i.vBarycentric.x +
                        i.vPositionOS_vert1 * i.vBarycentric.y +
                        i.vPositionOS_vert2 * i.vBarycentric.z;
   float3 vNormalOS   = i.vNormalOS_vert0   * i.vBarycentric.x +
                        i.vNormalOS_vert1   * i.vBarycentric.y +
                        i.vNormalOS_vert2   * i.vBarycentric.z;
   float3 vTangentOS  = i.vTangentOS_vert0  * i.vBarycentric.x +
                        i.vTangentOS_vert1  * i.vBarycentric.y +
                        i.vTangentOS_vert2  * i.vBarycentric.z;
   float3 vBinormalOS = i.vBinormalOS_vert0 * i.vBarycentric.x +
                        i.vBinormalOS_vert1 * i.vBarycentric.y +
                        i.vBinormalOS_vert2 * i.vBarycentric.z;

   // Interpolated texture coordinates:
   o.vUV = i.vUV_vert0 * i.vBarycentric.x + i.vUV_vert1 * i.vBarycentric.y +
           i.vUV_vert2 * i.vBarycentric.z;

   // Displace vertex by object's displacement map
   float fDisplacement = tDisplacement.SampleLevel( sPointClamp, o.vUV, 0 ).r;
         fDisplacement = fDisplacement * fDisplacementScale + fDisplacementBias;

   vPositionOS = vPositionOS + fDisplacement * vNormalOS;
```

```
  // Convert position and tangent frame from object space to world space by rotating and
  // translating because we are always rotating about y, we can simplify the math
  // somewhat for extra perf
  float3 vPositionWS = Rotate2D( vDir, vPositionOS ) + vInstancePosWS;
  float3 vNormalWS   = Rotate2D( vDir, vNormalOS );
  float3 vTangentWS  = Rotate2D( vDir, vTangentOS );
  float3 vBinormalWS = Rotate2D( vDir, vBinormalOS );

  o.vPositionSS = mul( mVP, float4( vPositionWS, 1 ) );
  o.vPositionWS = float4( vPositionWS, 1 );
  o.vNormalWS   = vNormalWS;
  o.vTangentWS  = vTangentWS;
  o.vBinormalWS = vBinormalWS;

  return o;
}
```

***Listing 7.*** *Example simple evaluation shader for rendering instanced tessellated characters.*

Given that we are rendering our character with displacement map, we must make a note about lighting. Traditionally, animated characters are rendered and lit using tangent-space normal maps (TSNM). However, there exists a concern with regards to using displacement mapping when lighting using tangent-space normal maps. In that case, we are essentially generating a new tangent frame as we are displacing, changing the actual displaced normal (as shown in Figure 12).

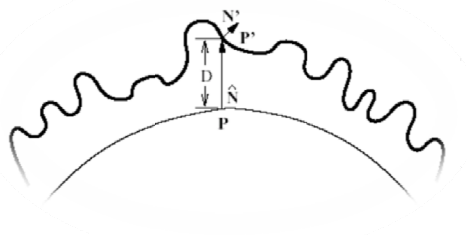


***Figure 12****. Displacement of the vertex modifies the normal used for rendering.* **P** *is the original point displaced in the direction of geometric normal* $\hat{N}$ *displacement amount* D. *The resulting point* **P'** *needs to be shaded using normal* $N'$.

However, we intend to light the displaced surface using tangent space normal map, using the encoded normals. In order to combine TSNM with displacement mapping, we need to ensure several constraints are met. Namely, that we are computing tangent space during rendering in the same exact manner as was used to generate the displacement and normal maps, and that the generation process for displacement and normal maps also used identical tangent space and models. In other words, ideally, the displacement map must be generated at the same time as the normal map. In that case, the normal encoded in the tangent-space normal map, would match the desired normal $N'$. By using publicly available AMD GPUMeshMapper tool which provides source code for tangent-space generation, we can ensure these requirements are met.

### 3.5.3 Tessellated Characters Level of Detail Control

In the Froblins demo, we use a three-level static LOD scheme, as discussed in section 3.3.2. Tessellation and displacement mapping are applied only to the characters in the most detailed level. In order to guarantee a stable frame rate in dense crowd situations, we compute the tessellation level as a function of the number of tessellated characters. This is effective in avoiding a polygonal count explosion and retains the performance benefits of geometry instancing. The tessellation level is set as follows:

$$T = clamp(\frac{2T_{max}}{N}, 1, T_{max})$$

Here, *T* is the tessellation level to be used for character instances in the first detail level, *N* is the number of character instances in the first detail level, and $T_{max}$ is the maximum tessellation level to use for a single character. This scheme effectively bounds the number of triangles created by the tessellator, and ensures that the primitive count will never increase by more than the cost of two fully tessellated characters. If there are more than two such characters in the view frustum, this scheme will divide the tessellated triangles evenly among them. Naturally, this can lead to popping as the size of the crowd changes dramatically from one frame to the next, but in a lively scene with numerous animated characters, this popping is hard to perceive.

### 3.5.3 Rendering Optimizations

Because hardware tessellation can generate millions of additional triangles, it is essential to minimize the amount of per-vertex computation. Our character vertex shaders already use a fairly expensive technique for skinned animation on the GPU (see Section 3.4), and performing these animation calculations inside the evaluation shader is wasteful.

We improve performance with a multi-pass approach for rendering out animated characters. We compute control cage pre-pass, where we can compute all relevant computations for the original low resolution mesh, such as animation and vertex transformations. This method is general and takes advantage of Direct3D® 10 *stream out* functionality. Note that in order to reduce the amount of memory being streamed out per character, as well as reduce vertex fetch and vertex cache reuse for the evaluation shader we augmented our control cage multi-pass method with vertex compression and decompression described below. Note that using this multi-pass method for control cage rendering is beneficial not only for rendering tessellated characters, but for any rendering pipeline where we wish to reuse results of expensive vertex operations multiple times. For example, we can use the results of the first pass for our animated and transformed characters for rendering into shadow maps and cube maps for reflections.

We perform the skinning calculations only once per character, on the super-primitive vertices, and the results are simply interpolated from the super-primitives by the evaluation shader. In the first pass, we render all of the character vertices an as instanced set of point primitives, perform skinning on them (as described in section 3.4), and stream out the results into a buffer. In the second (tessellated) pass, the instance ID and super-primitive vertex IDs are used by the evaluation shader to fetch the transformed

vertex data, interpolate a new vertex, and apply displacement mapping. Note that the only quantities that need to be output in the first pass are quantities affected by the transformations (such as vertex positions and normals, but not the texture coordinates or vertex colors, for example).

Although it is helpful to stream and re-use the skinning calculations, this alone is not very effective, because the vertex data will be streamed at full precision, and the evaluation shader must still pay a large cost in memory bandwidth and fetch instructions in order to retrieve it. Additionally, we would need to allocate sufficient stream out buffer to store transformed vertices. We use a compression scheme to pack the transformed vertices into a compact 128-bit format in order to remove this bottleneck and to reduce the associated memory footprint. This allows the tessellation pass to load a full set of transformed vertex data using a single fetch per super-primitive vertex. Although the compression scheme requires additional ALU cycles for both compression and decompression, this is more than paid for by the reduction in memory bandwidth and fetch operations in the evaluation shader.

We compress vertex positions by expressing them as fixed-point values which are used to interpolate the corners of a sufficiently large bounding box which is local to each character. The number of bits needed depends on the size of the model and the desired quality level, but it does not need to be extremely large. In our case, the dynamic range of our vertex data is roughly 600 cm. A 16-bit coordinate on this scale gives a resolution of about 90 microns, which is slightly larger than the diameter of a human hair.

We can compress the tangent frame by converting the basis vectors to spherical coordinates and quantizing them. Spherical coordinates are well suited to normal compression since every compressed value in the spherical domain corresponds to a unique unit-length vector. In a Cartesian representation (such as the widely used `DEC3N` format), a large fraction of the space of compressed values will go unused. What this means in practice is that a much smaller bit count can be used to represent spherical coordinates at a reasonable level of accuracy. We have found that using an 8-bit spherical coordinate pair for normals results in rendered images that are comparable in quality to a 32-bit Cartesian format. The main drawback of using spherical coordinates is that a number of expensive trigonometric functions must be used for compression and decompression, but we have found that the benefits of a small compressed format outweigh the additional ALU cost on current graphics hardware.

Texture coordinates are compressed by converting the UV coordinates into a pair of fixed-point values, using whatever bits are left. In order to ensure acceptable precision, this requires that the UV coordinates in the model be confined to the 0-1 range, with no explicit tiling of textures by the artist. For small textures, a smaller bit count could be used for the UV coordinates, provided that the UVs are snapped to the texel centers.

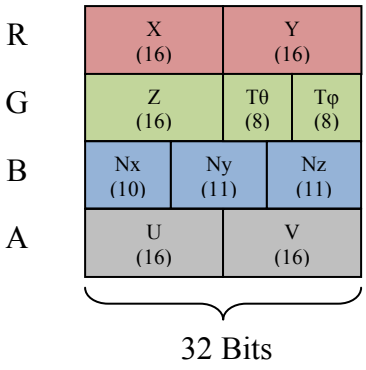


***Figure 13.*** *Data format used for compressed, animated vertices*

Our bit layout for the compressed vertices is shown in Figure 13, and corresponding compression and decompression code is shown in Listings 8 and 9. We use 16 bits for each component of the position, two 8-bit spherical coordinates for the tangent, 32 bits for the normal, and 16 for each UV coordinate. Since our tangent frames are orthogonal, we refrain from storing the binormal, and instead re-compute it based on the decompressed normal and tangent. Since a full 32-bit field is available, we use `DEC3N`-like compression for the normal, which requires fewer ALU operations than spherical coordinates. If additional data fields are needed, we have also found that 8-bit spherical coordinates can be used for the normal, at a quality level comparable to `DEC3N`. We experimented with all of these alternatives on the ATI Radeon™ HD 4870 GPU, but found little practical difference in performance or quality between any of them.

We believe that the compressed format that we use here would also make an excellent storage format for static geometry. In this case, (and also for the case of non-instanced characters) the decompression could be accelerated by leveraging the vertex fetch hardware to perform some of the integer to float conversions. We cannot do this in our case, because we must explicitly fetch vertex data with buffer loads, using the instance ID of the character, instead of using the fixed function vertex fetch. We obtained a 25% performance improvement via our multi-pass technique, and we observed gains as high as 37% with using the compression scheme. Due to quantization used for compression, there are subtle differences between the two images due to compression. However, these artifacts are difficult to notice in a dense, dynamic crowd of animated characters, and even difficult to discern in static comparison screenshots.

```
// Quantizes a floating point value (0-1) to a certain number of bits
uint Quantize( float v, uint nBits )
{
   float fMax = ((float) (1 << nBits))-1.0f;
   return uint( round(v*fMax) );
}

uint PackShorts( uint nHigh, uint nLow )
{
   return (nHigh << 16) | (nLow);
}

uint PackBytes( uint nHigh, uint nLow )
{
   return (nHigh << 8) | (nLow);
}

/// Converts a vector to spherical coordinates.
/// Theta (x) is in the 0-PI range.  Phi (y) is in the -PI,PI range
float2 CartesianToSpherical( float3 cartesian )
{
   cartesian = clamp( normalize( cartesian ), -1,1 ); // beware of rounding error
   float theta = acos( cartesian.z );
   float s     = sqrt( cartesian.x * cartesian.x + cartesian.y * cartesian.y );
   float phi   = atan2( cartesian.x / s, cartesian.y / s );
   if( s == 0 )
     phi = 0; // prevent singularity if normal points straight up

   return float2( theta, phi );
}

// Converts a normal vector to quantized spherical coordinates
uint2 CompressVectorQSC( float3 v, uint nBits )
{
   float2 vSpherical = CartesianToSpherical( v );

   return uint2( Quantize(  vSphericalNorm.x / PI, nBits ),
                 Quantize( (vSphericalNorm.y + PI ) / ( 2*PI ), nBits ) );
}

// Encodes position as fixed-point lerp factors between AABB corners
uint3 CompressPosition( float3 vPos, float3 vBBMin, float3 vBBMax, uint nBits )
{
   float3 vPosNorm = saturate( (vPos - vBBMin) / (vBBMax-vBBMin) ):
   return uint3( Quantize( vPosNorm.x, nBits ),
                 Quantize( vPosNorm.y, nBits ),
                 Quantize( vPosNorm.z, nBits ) );
}

uint PackCartesian( float3 v )
{
   float3 vUnsigned = saturate( (v.xyz * 0.5) + 0.5 );
   uint nX = Quantize( vUnsigned.x, 10 );
   uint nY = Quantize( vUnsigned.y, 11 );
   uint nZ = Quantize( vUnsigned.z, 11 );
   return ( nX << 22 ) | ( nY << 11 ) | nZ;
}

uint4 PackVertex( CompressedVertex v, float3 vBBoxMin, float3 vBBoxMax )
{
   uint3 nPosition  = CompressPosition( v.vPosition, vBBoxMin, vBBoxMax, 16 );
   uint2 nTangent   = CompressVectorQSC( v.vTangent, 8 );

   uint4 nOutput;
   nOutput.x = PackShorts( nPosition.x, nPosition.y );
   nOutput.y = PackShorts( nPosition.z, PackBytes( nTangent.x, nTangent.y ) );
   nOutput.z = PackCartesian ( v.vNormal );
   nOutput.w = PackShorts( Quantize( vUV.x, 16 ), Quantize( vUV.y, 16 ) );
   return nOutput;
}
```

***Listing 8.*** *Compression code for vertex format given in Figure 13.*

```
float DeQuantize( uint n, uint nBits )
{
   float fMax = ((float) (1 << nBits)) - 1.0f;
   return float(n)/fMax;
}

float3 DecompressVectorQSC( uint2 nCompressed, uint nBitCount )
{
   float2 vSph = float2( DeQuantize( nCompressed.x, nBitCount ),
                         DeQuantize( nCompressed.y, nBitCount ) );
   vSph.x = vSph.x * PI;
   vSph.y = (2 * PI * vSph.y) - PI
   float fSinTheta  = sin( vSph.x );
   float fCosTheta  = cos( vSph.x );
   float fSinPhi    = sin( vSph.y );
   float fCosPhi    = cos( vSph.y );
   return float3(  fSinPhi * fSinTheta, fCosPhi * fSinTheta, fCosTheta );
}

float3 DecompressPosition( uint3 nBits, float3 vBBMin, float3 vBBMax, uint nCount )
{
   float3 vPosN = float3( DeQuantize( nBits.x, nCount),
                          DeQuantize( nBits.y, nCount),
                          DeQuantize( nBits.z, nCount) );
   return lerp( vBBMin.xyz, vBBMax.xyz, vPosN );
}

float3 UnpackPosition( uint4 nPacked, float3 vBBoxMin, float3 vBBoxMax )
{
   uint3 nPos;
   nPos.xy = uint2( nPacked.x >> 16, nPacked.x & 0x0000ffff );
   nPos.z  = nPacked.y >> 16;
   return DecompressPosition( nPos, vBBoxMin, vBBoxMax, 16 );
}

float2 UnpackUV( uint4 nPacked )
{
   uint2 nUV  = uint2( nPacked.w >> 16, nPacked.w & 0x0000ffff );
   float2 vUV = float2( DeQuantize( nUV.x, 16 ), DeQuantize( nUV.y, 16 ) );
   return vUV;
}

float3 UnpackTangent( uint4 nPacked )
{
   uint2 nTan = uint2( (nPacked.y >> 8) & 0xff, nPacked.y & 0xff );
   return DecompressVectorQSC( nTan, 8 );
}

float3 UnpackCartesian( uint n )
{
   uint  nX = (n >> 22) & 0x3FF;
   uint  nY = (n >> 11) & 0x7FF;
   uint  nZ = n         & 0x7FF;
   float fX = (2.0f * DeQuantize( nX, 10 )) - 1.0f;
   float fY = (2.0f * DeQuantize( nY, 11 )) - 1.0f;
   float fZ = (2.0f * DeQuantize( nZ, 11 )) - 1.0f;
   return  float3( fX, fY, fZ );
}

CompressedVertex UnpackVertex( uint4 nPacked, float3 vBBoxMin, float3 vBBoxMax )
{
   CompressedVertex vVert;
   vVert.vPosition = UnpackPosition( nPacked, vBBoxMin, vBBoxMax );
   vVert.vNormal   = UnpackCartesian( nPacked.z );
   vVert.vTangent  = UnpackTangent( nPacked );
   vVert.vBinormal = normalize( cross( vVert.vTangent, vVert.vNormal ) );
   vVert.vUV       = UnpackUV( nPacked );
   return vVert;
}
```

***Listing 9.*** *Decompression code for vertex format given in Figure 13*

### 3.5.4 Displacement Map Tips and Ensuring Watertightness

We would like to share several practical tips for generation and using of displacement maps that we've learned throughout our process. Firstly, the method used for generation of displacement maps must match the method for evaluating subdivided surface. This naturally correlates to the absolute need to know the process used by the modeling tool used for map generations. Many DCC tools such as Autodesk Maya® will first use approximating subdivision process, such as Catmull-Clark subdivision method, on the control mesh (the low resolution, or super-primitive, mesh). Once the mesh has been smoothed, then the fine-scale details are captured into a scalar or vector displacement map. When using these tools, we must evaluate the final surface using Catmull-Clark subdivision methods. However, the evaluation shaders are reasonably expensive, which precipitated our decision to use interpolative planar subdivision due to its extreme simplicity for evaluation. Additionally a number of concerns arise with topology fix-up and treatment of extraordinary points, as well as patch reordering to ensure watertightness during displacement. However, should the interested reader may wish to investigate further about using GPU tessellation for Catmull-Clark surfaces, they can find additional material and further references in [TATARCHUK08].

In our case, we used the AMD GPUMeshMapper tool ([AMDGMM08]), designed specifically for robust generation of displacement maps for interpolative planar subdivision. Specifically, given a pair of low and high resolution meshes, this tool provides a number of different options for controlling the envelopes for ray casting from low to high resolution map in order to capture displacement and normal information. Furthermore, in order to achieve controllable results at run-time, we must know the exact floating point values for displacement scale and bias for the generated displacement map. This tool provides this information, collected during the generation process, in the form of parameters which can be used directly in the shader.

Particular care needs to be paid during displacement mapping in order to generate resulting watertight surfaces. This is true regardless of the subdivision method used for evaluation. One challenge with rendering complex characters with displacement maps that contain texture *uv* borders is the introduction of texture *uv* seams (see Figure 14 for an example of such a displacement map). Unless neighboring *uv* borders are laid out with the same orientations and lengths, displacing with these maps will introduce geometry cracks along the *uv* borders (Figure 14). This happens due to bilinear discontinuities and to varying floating point precision on different regions of the texture map. Seamless parameterizations remove bilinear artifacts, but do not solve floating point precision issues. One-to-one parameterization is extremely difficult to obtain for complex characters, if not impossible in practical scenarios. We solve this problem during the map generation process, rather than at run-time, via additional features implemented as part of the GPUMeshMapper tool. We post-process our displacement maps by correcting all the texture *uv* borders during the displacement map generation, by identifying the border triangle edges and performing filtering across edges (with additional fix-up) to alleviate displacement seams.

*Displacement map*

*Rendered character using the displacement map with a seam*

*Zoomed-in view shows a crack in the surface*

***Figure 14.*** *Example of a visible crack generated due to inconsistent values across the edges of displacement map for this character. On the left we highlighted the specific edges along the seam. Note that the adjacent edges for this seam do not have uniform parameterization.*

## 3.6 Lighting and Shadowing

### 3.6.1 Rendering Shadows in Large Scale Environment

High quality rendering system requires dynamic shadows cast by characters onto the environment and themselves. To manage shadow map resolution, our system implements *Parallel Split Shadow Maps* [ZSXL06]. The view-frustum test described in Section 3.3 is used to ensure that only characters that are within a particular parallel split frustum are rendered. Occlusion culling could also be used for shadow maps as well, but we do not do this in our system, because only characters and smaller scene elements are rendered into the shadow maps and there is little to cull the characters against (shadows cast by terrain are handled separately). We use aggressive filtering for generation of soft shadows. This allows us to use further mesh simplification for the LOD rendered into shadow maps. For characters in the higher-detail shadow frusta, we use the same simplified geometry that is used for the most distant level of detail during normal rendering. For more distant shadows, we can use a more extreme simplification.

### 3.6.2 Lighting

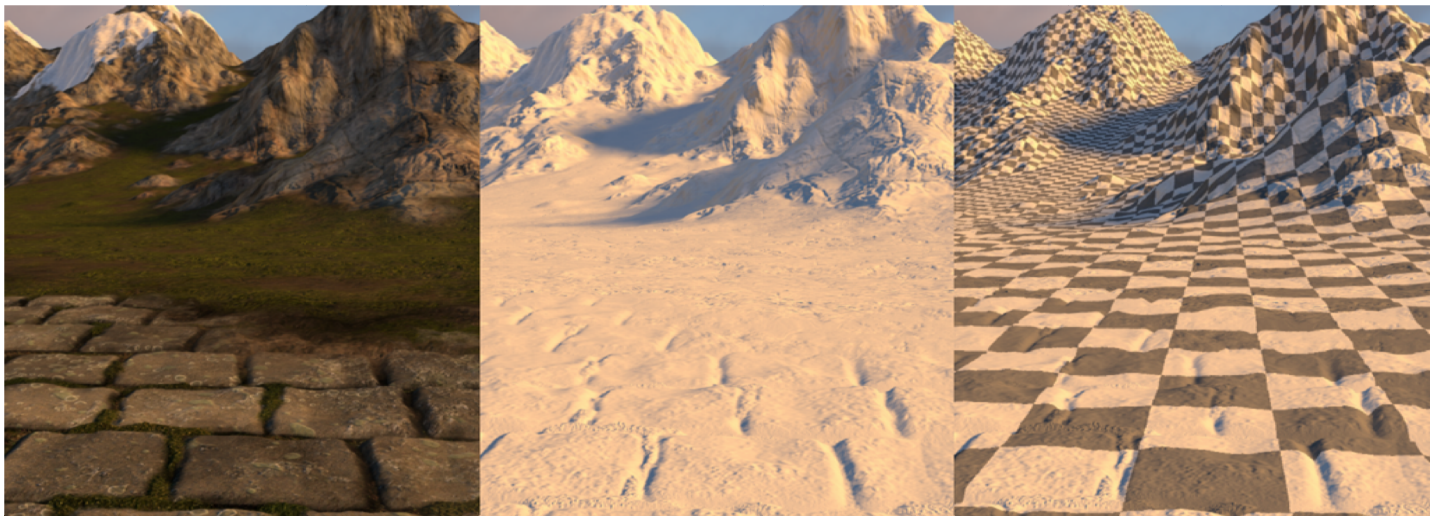

***Figure 15.*** *A medium resolution spherical harmonic light map is used to light a highly detailed terrain: [left] the fully shaded terrain, [center] just the lighting from the spherical harmonic light map, [right] a checkerboard pattern overlay to indicate light map texel density.*

In this section we describe the lighting system used in the AMD *Froblins* demo. The demo does not have a dynamic day/night cycle, our global scene lighting is static which enables us to use a light map to store precomputed incident lighting on the terrain. We chose to use a Spherical Harmonic Light Map (SHLM) [CHEN08]. A single texel in an SHLM stores a complete lighting environment at that point. At run time, the SHLM is queried and the lighting environment is evaluated in the direction of the shading normal, for example, to compute a diffuse lighting term. Because the shading normal is decoupled from the light map, a SHLM can be stored at a lower resolution than the finest level of shading detail while still providing detailed lighting results (Figure 15). This decoupling also enables us to use the terrain's static light map like a radiance cache for computing lighting on scene elements such as our dynamic characters and props. We briefly motivate and discuss our method for generating the SHLM and then show how we used this data for lighting. Finally, we present a simple technique for integrating dynamic shadows, cast by our characters, into the static scene while avoiding "double shadowing" artifacts in regions where dynamic character shadows overlap static terrain shadows.

#### 3.6.2.1 *SHLM Generation*

Our outdoor scene is comprised of two main global light emitters. The primary emitter is the sun which is modeled as a directional light source. The secondary emitter is the sky itself which is modeled using a high dynamic range environment map. To generate the SHLM, the terrain is divided uniformly into a grid that matches our desired light map resolution of 1024 × 1024. At the center of each grid square, lighting samples are taken at a point just above the terrain. Samples are taken at a distance off the terrain of approximately half of our character's height, as shown in Figure 16. This height was chosen to ensure that the samples would work well for lighting both the terrain as well as the characters. Samples taken on the ground only capture a partial lighting environment and are poorly suited for shading points above the

terrain that may require the missing lower hemisphere of the lighting environment. On the other hand, samples taken too far off the ground can create artifacts when used for shading the terrain. In particular, this can lead to missing or unnatural terrain shadow boundaries as well as incorrect self-reflectance. In practice, we found that taking samples at a moderate height above the terrain we were able to capture complete lighting environments that were useful for shading both the terrain and the characters.

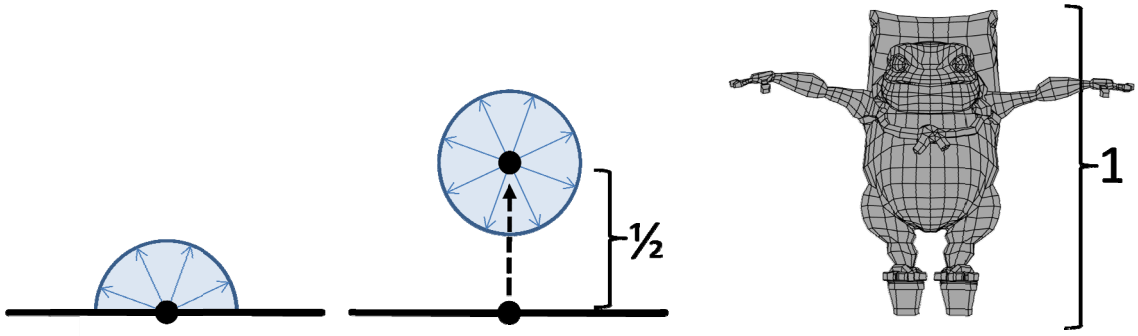


***Figure 16.*** *The lighting environment is captured at a point by firing rays into the environment. [Left] A sample taken on the terrain surface does not contain useful lighting data in the lower hemisphere and is not suitable for shading characters that may have shading normals that point down into the lower hemisphere of the lighting environment. [Center] A sample point is offset from the terrain at a distance of approximately half of our characters' height, ensuring the captured lighting environment is useful for shading characters as well as the terrain itself. [Right] A character requires a full spherical lighting environment that captures bounced lighting from the ground below.*

At each sample point, direct and indirect light is captured and projected into spherical harmonics. The direct light from the sun is computed by casting a distribution of rays in the direction of the sun and testing for occlusion. Indirect light from the sun and from the sky is collected by firing 800 rays, with a modest recursion limit, in all directions on the sphere, using a stratified sampling scheme. The incident radiance is then stored using 3$^{rd}$ order spherical harmonics. This data is written to disk as 16-bit floating point textures using the OpenEXR image file format.

### 3.6.2.2 *Rendering Using SHLM*

For rendering, we pack the 3$^{rd}$ order spectral SHLM data into seven RGBA16F textures. Others have suggested using various compression schemes [WWS*07] [Hu08] but our memory budget did not require us to compress the data and we found that the uncompressed coefficients gave slightly higher quality lighting results particularly in areas of high contrast such as shadow boundaries. In these scenarios we found that we could get higher quality lighting results by storing a lower resolution uncompressed SHLM rather than a higher resolution compressed SHLM. We did observe that the DC components of the SHLM could be stored using a shared exponent texture format (RGBE) with minimal loss in quality; higher order spherical harmonic coefficients cannot be stored in this way because the format does not allow for negative values.

Final lighting is computed in a pixel shader by sampling the SHLM, removing the dominant directional light from the linear terms, then summing the contribution of this dominant directional light and the "residual" environment lighting [SLOAN08]. DirectX® HLSL shader code demonstrating this is provided in the listing at the end of this section.

Since our characters are dynamic, their shadows cannot be baked in to the SHLM as a preprocess. Instead, a more traditional real-time shadowing method, parallel-spit shadow mapping [ZSXL06], was used to render their shadows. We did not want to simply darken the terrain wherever a character casts a shadow, this would incur a double shadowing artifact in regions that are already shadowed in the light map due to terrain self occlusion as shown in Figure 17. Ideally we would like the shadow map to only attenuate the sun's contribution to the light map. This can be very nicely approximated by separating a dominant directional light from the lighting environment in the terrain's pixel shader. Please refer to [SLOAN08] for a discussion on extracting a dominant light from a spherical harmonic lighting environment.

Once the dominant directional light is removed from the lighting environment sampled from the SHLM, we test if this light's direction corresponds with the sun's direction. If both vectors point in the same direction then we determine that the pixel is in direct sun light and the shadow map should be applied. If the vectors disagree, then the pixel is considered to already be occluded from the sun and thus the effects of the shadow map are faded out. Once the adjusted shadowing term is computed, it is then used to attenuate the dominant lighting term which is then added to the remaining spherical harmonic lighting environment. Please see the sample code provided at the end of this section.

***Figure 17.*** *Characters cast shadows on the terrain. The left half of the image is in the shadow of a mountain, the right half is in direct sun light. [Top] Characters incorrectly cast double shadows on occluded region of the terrain. [Bottom] A shadow correction factor is applied to prevent double shadowing artifacts.*

We also use the terrain's SHLM for lighting our characters and scene props (Figure 18). In the characters' pixel shader, the point being shaded is projected onto the texture space of the SHLM and samples are taken which are then used to approximate a lighting environment for the character. This does not provide any lighting variation along the vertical axis but in practice it works quite well even for tall scene elements such as the tent in the figure's foreground or the pagoda in the figure's background. Additional texture maps could be used to store vertical gradients for the spherical harmonic coefficients; this would provide more accurate lighting environment reconstruction for points located above the terrain [AKDS04].

***Figure 18.*** *Dynamic characters (bottom) and other static scene props (top) build an approximate lighting environment for shading by sampling from the terrain's SHLM.*

```
// Samplers
sampler g sSHLMPoint;
sampler g sSHLMBilinear;
sampler g sSHLMTrilinear;
sampler g sSHLMAnisotropic;

// SHLM Textures
Texture2D<float4> tSHLM R0; // DC & linear (red)
Texture2D<float4> tSHLM R1; // first 4 quadratic (red)
Texture2D<float4> tSHLM G0; // DC & linear (green)
Texture2D<float4> tSHLM G1; // first 4 quadratic (green)
Texture2D<float4> tSHLM_B0; // DC & linear (blue)
Texture2D<float4> tSHLM_B1; // first 4 quadratic (blue)
Texture2D<float3> tSHLM RGB2;

// Global parameters, set by the application
float2 g vSHLMEnvironmentScale;
float3 g_vSHLMSunDirectionWS;

// ============================================================================
// Evaluate a SH basis functions for a given direction
// ============================================================================
void SHEvalDirection ( float3 vDirection, out float4 vOut[3] )
{
   float3 vDirection2 = vDirection * vDirection;

   vOut[0].x = 0.282095;

   vOut[0].y = -0.488603 * vDirection.y;
   vOut[0].z = 0.488603 * vDirection.z;
   vOut[0].w = -0.488603 * vDirection.x;

   vOut[1].x = 1.092548 * vDirection.x * vDirection.y;
   vOut[1].y = -1.092548 * vDirection.y * vDirection.z;
   vOut[1].z = 0.315392 * (3.0*vDirection2.z - 1.0);
   vOut[1].w = -1.092548 * vDirection.x * vDirection.z;
   vOut[2].x = 0.546274 * (vDirection2.x - vDirection2.y);

   // Last three channels go unused
   vOut[2].yzw = 0.0;

   return;
}

// ============================================================================
// Turn world space position into light map UV
// ============================================================================
float2 ComputeLightMapUV ( float3 vPositionWS )
{
   float2 vUV = (vPositionWS.xz / g vSHLMEnvironmentScale) + 0.5;
   return vUV;
}

// ============================================================================
// Assemble the SH coefficients & Dominant light info. SH is 3rd order (9
// coefficients per color channel). Coefficients are stored in an array of
// float4 vectors, the last the components of the last float4 vector in each
// array go unused.
//
// Inputs:
// vUV - Texture coord
// sLightMapSampler - sampler state
//
// Outputs:
// vSHr[] - Residual lighting environment (dominant light removed)
// vSHg[] - Residual lighting environment (dominant light removed)
// vSHb[] - Residual lighting environment (dominant light removed)
// cDominantColor - Dominant directional light color
// vDominantDir - Dominant directional light direction
// ============================================================================
void GetLightingEnvironment ( float2 vUV, sampler sLightMapSampler,
                              out float4 vSHr[3], out float4 vSHg[3],
```

```
                                out float4 vSHb[3], out float3 cDominantColor,
                                out float3 vDominantDir )
{
   vSHr[0] = tSHLM_R0.Sample( sLightMapSampler, vUV ); // DC & linear terms
   vSHg[0] = tSHLM_G0.Sample( sLightMapSampler, vUV ); // DC & linear terms
   vSHb[0] = tSHLM_B0.Sample( sLightMapSampler, vUV ); // DC & linear terms

   vSHr[1] = tSHLM_R1.Sample( sLightMapSampler, vUV ); // first 4 quadratic
   vSHg[1] = tSHLM_G1.Sample( sLightMapSampler, vUV ); // first 4 quadratic
   vSHb[1] = tSHLM_B1.Sample( sLightMapSampler, vUV ); // first 4 quadratic

   // final quadratic (red, green, blue)
   float3 vTmp = tSHLM_RGB2.Sample( sLightMapSampler, vUV );
   vSHr[2].x = vTmp.r; // last 3 channels of vSHr[2] go unused
   vSHg[2].x = vTmp.g; // last 3 channels of vSHr[2] go unused
   vSHb[2].x = vTmp.b; // last 3 channels of vSHr[2] go unused

   // extract dominant light direction from linear SH terms
   vDominantDir = (vSHr[0].yzw * 0.3 + vSHg[0].yzw * 0.59 + vSHb[0].yzw*0.11);
   vDominantDir = normalize( float3(-vDominantDir.zx, vDominantDir.y) );

   // turn dom direction into an SH directional light with unit intensity
   float4 Ld[3];
   SHEvalDirection( vDominantDir, Ld );
   Ld[0] *= 2.95679308573; // factor to make it unit intensity
   Ld[1] *= 2.95679308573;
   Ld[2] *= 2.95679308573;

   float fDenom = dot(Ld[0],Ld[0])+dot(Ld[1],Ld[1])+(Ld[2].x*Ld[2].x);

   // find the color of the dominant light
   cDominantColor.r =
      (dot(Ld[0],vSHr[0])+dot(Ld[1],vSHr[1])+(Ld[2].x*vSHr[2].x))/fDenom;

   cDominantColor.g =
      (dot(Ld[0],vSHg[0])+dot(Ld[1],vSHg[1])+(Ld[2].x*vSHg[2].x))/fDenom;

   cDominantColor.b =
      (dot(Ld[0],vSHb[0])+dot(Ld[1],vSHb[1])+(Ld[2].x*vSHb[2].x))/fDenom;

   // subtract dominant light from original lighting environment so we
   // don't get double lighting
   vSHr[0] = vSHr[0] - Ld[0]*cDominantColor.r;
   vSHg[0] = vSHg[0] - Ld[0]*cDominantColor.g;
   vSHb[0] = vSHb[0] - Ld[0]*cDominantColor.b;

   vSHr[1] = vSHr[1] - Ld[1]*cDominantColor.r;
   vSHg[1] = vSHg[1] - Ld[1]*cDominantColor.g;
   vSHb[1] = vSHb[1] - Ld[1]*cDominantColor.b;

   vSHr[2].x = vSHr[2].x - Ld[2].x*cDominantColor.r;
   vSHg[2].x = vSHg[2].x - Ld[2].x*cDominantColor.g;
   vSHb[2].x = vSHb[2].x - Ld[2].x*cDominantColor.b;
}
// ============================================================================
// Compute the amount of shadow to apply. fShadow comes from a shadow
// map lookup.
// ============================================================================
float ComputeDirectLightingShadowFactor ( float3 vDominantLightDir,
                                          float fShadow )
{
   // in order to avoid double darkening we figure out how much the dominant
   // light matches up with the actual directional light source and then use
   // that to figure out how much extra darkening we should apply.

   // thresholds for fading in shadow. the cosine of the angle between the
   // two vectors is mapped to the [0,1] range. these thresholds mark the
   // points within that threshold that the shadow is faded in.  Tweak these
   // to change the range over which the shadow is faded in/out.
   static const float fShadowStart = 0.45; // start fading at ~63 degrees
   static const float fShadowStop  = 0.95; // full shadow at ~18 degrees
```

```
   // smoothstep to fade in/out shadow. Dot product is scaled/biased
   // from [-1,1] into [0,1] range. threshold terms determine where
   // the fade in/out boundaries are. we call this "exposure to sun"
   // because it approximates how exposed you are to the sun and
   // thus how much shadow should be allowed.
   float fAngle = dot(vDominantLightDir,g_vSHLMSunDirectionWS)*0.5+0.5;
   float fExposureToSun = smoothstep(fShadowStartThreshold,
                                     fShadowStopThreshold,
                                     fAngle);
   // amount of dominant light to remove
   float fPercentShadowed = lerp( 1, fShadow, fExposureToSun);

   return fPercentShadowed;
}
// ============================================================================
// Compute shadowed diffuse lighting.  We pass the dominant light direction
// and dominant light color back to the caller so that it may be used for
// specular/glossy calculations. The adjusted shadow factor is passed back
// in the alpha channel of the returned vector so that it may be used for
// shadowing any specular/glossy shading terms that the caller computes.
// ============================================================================
float4 ComputeDiffuse ( float3 vPositionWS, float3 vNormalWS, float fShadow,
                        out float3 vDominantLightDir,
                        out float3 cDominantLightColor )
{
   // compute a texture coord for the light map
   float2 vUV = ComputeLightMapUV( vPositionWS );

   // get the lighting environment
   float4 vSHLightingEnvR[3], vSHLightingEnvG[3], vSHLightingEnvB[3];
   GetLightingEnvironment( vUV, g_sSHLMBilinear,
                           vSHLightingEnvR, vSHLightingEnvG, vSHLightingEnvB,
                           cDominantLightColor, vDominantLightDir );

   // build basis for lambertian reflectance function
   float4 vSHLambert[3];
   SHEvalDirection( vNormalWS, vSHLambert );

   // the lambertian SH convolution coefficients for the first three bands
   float3 vConvolution = float3( 1.0, 2.0/3.0, 1.0/4.0 );
   vSHLambert[0] *= vConvolution.xyyy;
   vSHLambert[1] *= vConvolution.zzzz;
   vSHLambert[2].x *= vConvolution.z;

   // apply shadow to the direct dominant light
   float fShadowFactor =
      ComputeDirectLightingShadowFactor( vDominantLightDir, fShadow );

   cDominantLightColor *= fShadowFactor;

   // direct diffuse lighting (from dominant directional light)
   float3 cDiffuse =
      max( 0, dot(vNormalWS,vDominantLightDir) ) * cDominantLightColor;

   // diffuse light from lighting environment (dominant light removed)
   cDiffuse.r += dot( vSHLambert[0], vSHLightingEnvR[0] ); // DC & linear
   cDiffuse.g += dot( vSHLambert[0], vSHLightingEnvG[0] );
   cDiffuse.b += dot( vSHLambert[0], vSHLightingEnvB[0] );
   cDiffuse.r += dot( vSHLambert[1], vSHLightingEnvR[1] ); // quadractic
   cDiffuse.g += dot( vSHLambert[1], vSHLightingEnvG[1] );
   cDiffuse.b += dot( vSHLambert[1], vSHLightingEnvB[1] );
   cDiffuse.r += vSHLambert[2].x * vSHLightingEnvR[2].x;
   cDiffuse.g += vSHLambert[2].x * vSHLightingEnvG[2].x;
   cDiffuse.b += vSHLambert[2].x * vSHLightingEnvB[2].x;

   cDiffuse = max( 0, cDiffuse );
   return float4(cDiffuse, fShadowFactor);
}
```

***Listing 10.*** *HLSL shader code implementing the spherical harmonic light map techniques described in this section.*

## 3.7 Conclusion

In this chapter we covered various aspects of simulating and rendering large crowds of autonomous characters using the massive parallelization available on the latest commodity GPUs. We described methods for computing dynamic path finding, using global model and local avoidance for handling character-to-character collisions. In our large-scale environment with thousands of highly detailed, intelligent characters, the *Froblins* (frog goblins), are concurrently simulated, animated and rendered entirely on the GPU. The *Froblins* demo contains 3000 characters, rendering at various levels of details, ranging from coarsest level at only 900 polygons all the way to over 1.6M triangles at extreme close-ups. We render thousands of animated intelligent characters from a variety of viewpoints ranging from extreme close-ups to far away "bird's eye" views of the entire system. Our system combines state-of-the-art parallel artificial intelligence computation for dynamic pathfinding and local avoidance on the GPU, massive crowd rendering with LOD management with high end rendering capabilities such as tessellation for high quality close-ups and stable performance, terrain system, cascaded shadows for large-range environments, and an advanced global illumination system. We are able to render our world at interactive rates (over 20 fps on ATI Radeon® HD 4870) with staggering polygon count (6 – 8 million triangles on average at 20-25 fps), while maintaining the full high quality lighting and shadowing solution.

## 3.8 Acknowledgements

We would like to thank all the creative, hard-working and over-all fun folks who contributed to this demo. Specifically, Abe Wiley, our lead artist, deserves a special mention for all his patience, diligence and painstaking attention to detail. The talented artists from Chaingun Studios and Exigent were also absolutely crucial to the success of this project, and we'd like to thank them for their contributions. We'd also like to thank the following folks from AMD Game Computing Group: Dan Abrahams-Gessel, Justin Hensley, Jason Yang and Raja Koduri, as well as AMD graphics driver developers who helped stabilize bleeding-edge advanced features, specifically, Tim Kelley and Matt Johnson. The GPU tools group was extremely helpful by working together on a tool specifically designed for robust generation of displacement maps with tessellation in mind, and we'd particularly like to thank Peter Lohrmann and Budirijanto Purnomo for their work on this tool.

## 3.9 References

[AHH08] AKENINE-MÖELLER, T., HAINES, E. AND HOFFMAN, N. 2008. *Real-Time Rendering*. 3rd ed. A.K. Peters, Ltd.

[AMDGMM08] AMD GPUMeshMapper. 2008. http://developer.amd.com/gpu/MeshMapper/Pages/default.aspx

[AKDS04] ANNEN, T., KAUTZ, J., DURAND, F., AND SEIDEL, H. P. 2004. Spherical Harmonic Gradients for Mid-Range Illumination. Rendering Techniques 2004: Eurographics Symposium on Rendering.

[CHEN08] CHEN, H. 2008. Lighting and Material of HALO 3. Game Developer's Conference (San Francisco).

[EVERITT01] EVERITT, C. 2001. Interactive Order-Independent Transparency. Technical report, NVIDIA Corporation.

[FIORINI SHILLER98] FIORINI, P., AND SHILLER, Z. 1998. Motion Planning in Dynamic Environments Using Velocity Obstacles. International Journal on Robotics Research 17(7), 760-772.

[GEE08] GEE, K. 2008. Direct3D® 11 Tessellation. Presentation. Gamefest 2008, Seattle, WA, July 2008.

[GKM93] GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical Z-buffer visibility. In SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques, ACM, New York, NY, USA, pp. 231–238.

[HARADA07] HARADA, T. 2007. Real-Time Rigid Body Simulation on GPUs. In GPU Gems 3, Nguyen, H. ed., Addison-Wesley.

[HU08] HU, Y. 2008. Lightmap Compression in HALO 3. Game Developer's Conference (San Francisco).

[JEONGWHITAKER07A] JEONG, W.-K, AND WHITAKER, R.T. 2007. A Fast Eikonal Equation Solver for Parallel Systems. SIAM conference on Computational Science and Engineering 2007, Technical Sketches

[JEONGWHITAKER07B] JEONG, W.-K, AND WHITAKER, R.T. 2007. A Fast Iterative Method for a Class of Hamilton-Jacobi Equations on Parallel Systems. University of Utah School of Computing Technical Report UUCS-07-010.

[KLEIN08] KLEIN, A. 2008. Introduction to the Direct3D® 11 Graphics Pipeline. Presentation. Gamefest 2008, Seattle, WA, July 2008.

[MHR07] MILLÁN, E., HERNÁNDEZ, B., AND RUDOMIN, I. 2007. Large Crowds of Autonomous Animated Characters Using Fragments Shaders and Level of Detail. ShaderX5 : Advanced Rendering Techniques. Engel, W. (Editor), Charles River Media, December 2006.

[SLOAN08] SLOAN, P. 2008. Stupid Spherical Harmonic (SH) Tricks. Game Developer's Conference (San Francisco).

[TATARCHUK08] TATARCHUK, N. 2008. Advanced Topics in GPU Tessellation: Algorithms and Lessons Learned. Presentation. Gamefest 2008, Seattle, WA, July 2008.

[TCP06] TREUILLE, A., COOPER, S., AND POPOVIĆ, Z. 2006. Continuum Crowds. *ACM Trans. Graph.* 25, 3 (Jul. 2006), pp. 1160-1168, Boston, MA.

[TSITSIKLIS95] TSITSIKLIS, J. N. 1995. Efficient Algorithms for Globally Optimal Trajectories. IEEE Transactions on Automatic Control 40, 9 (Sept.), 1528-1538.

[VBPS*08] VAN DEN BERG, J., PATIL, S., SEWALL, J., MANOCHA, D., AND LIN, M. 2008. Interactive Navigation of Multiple Agents in Crowded Environments. In Proceedings of the 2008 Symposium on interactive 3D Graphics and Games (Redwood City, California, February 15 - 17, 2008). SI3D '08. ACM, New York, NY, 139-147.

[WWS*07] WANG, L., WANG, X., SLOAN, P., WEI, L., TONG, X., AND GUO, B. 2007. Rendering from Compressed High Dynamic Range Textures on Programmable Graphics Hardware. ACM Symposium on Interactive 3D Graphics and Games.

[YCP*08] YEH, H., CURTIS, S., PATIL, S., VAN DEN BERG, J., MANOCHA, D., AND LIN, M. 2008. Composite Agents. In the proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2008.

[ZSXL06] ZHANG, F., SUN, H., XU, L., AND LUN, L. K. 2006. Parallel-split shadow maps for large-scale virtual environments. In VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications, ACM, New York, NY, USA, pp. 311–318.

# Appendix A

```
// Solve for roots of quadratic equation
float2 EvalQuadratic( float a, float b, float c )
{
   float2 roots;
   roots.x = (-b + sqrt( b*b-4*a*c ))/(2*a);
   roots.y = (-b - sqrt( b*b-4*a*c ))/(2*a);

   if( b*b <= 4*a*c )
   {
      roots = float2( INF-1, INF-1 );
   }

   return roots;
}

// Solve for the the potential of the current position based on the
// potential of the neighbors and the cost of moving here from there. Refer
// to Jeong "A Fast Eikonal Equation Solver for Parallel Systems" 2007.
float QuadraticSolver( float fPhiMx,  float fPhiMy,
                       float fCostMx, float fCostMy )
{
   float a = fPhiMx;
   float b = fPhiMy;
   float c = fCostMx;
   float d = fCostMy;

   float a1 = c * c + d * d;
   float b1 = -(2 * a * d * d + 2 * b * c * c);
   float c1 = a * a * d * d + b * b * c * c - c * c * d * d;

   float2 roots = EvalQuadratic( a1, b1, c1 );
   float fTmp = max( roots.x, roots.y );

   return fTmp;
}

float EvaluateFiniteDifference( float fPhi,  float fCost,
                                float4 vPhi, float4 vCost )
{
   float fPhiX, fPhiY, fCostX, fCostY;
   float fPhiN = vPhi[0], fPhiS = vPhi[1], fPhiW = vPhi[2],
                 fPhiE = vPhi[3];
   float fCostN = vCost[0], fCostS = vCost[1], fCostW = vCost[2],
                  fCostE = vCost[3];

   //====Calculate upwind direction for X====
   if( fPhiW < INF || fPhiE < INF )
   {
      // Figure out if west or east are "cheaper"
      if( fPhiW + fCostW <= fPhiE + fCostE )
      {
        fPhiX  = fPhiW;
        fCostX = fCostW;
      }
      else
      {
         fPhiX  = fPhiE;
         fCostX = fCostE;
       }
   }
```

***Listing 11.*** *HLSL code for iterative eikonal solver*

```
      //====Calculate upwind direction for Y====
      if( fPhiN < INF || fPhiS < INF )
      {
         bInvalidY = false;

         // Figure out if north or south are "cheaper"
         if( fPhiN + fCostN <= fPhiS + fCostS )
         {
            fPhiY  = fPhiN;
            fCostY = fCostN;
          }
          else
          {
             fPhiY  = fPhiS;
             fCostY = fCostS;
          }
      }

     //Save for new potential in this location by solving quadratic
     float result = 0;
     result = QuadraticSolver( fPhiX, fPhiY, fCostX, fCostY );
     result = min( min( fPhiY + fCostY, fPhiX + fCostX ), result );

     // Potential should only be decreasing
     result = ( result > fPhi ) ? fPhi : result;
   }

float4 EikonalSolverIteration()
{
   float4 vCurPhi = tPhiMap.SampleLevel( sPhiPoint, v.vUV, 0 );
   float4 vCurCost = tCostMap.SampleLevel( sCostPoint, v.vUV, 0 );

   // Fetch potential values. Fetches out of domain = INF
   float4 vPhiN = tPhiMap.SampleLevel( sPhiPoint, v.vUV, 0, int2( 0,-1 ) );
   float4 vPhiS = tPhiMap.SampleLevel( sPhiPoint, v.vUV, 0, int2( 0, 1 ) );
   float4 vPhiW = tPhiMap.SampleLevel( sPhiPoint, v.vUV, 0, int2(-1, 0 ) );
   float4 vPhiE = tPhiMap.SampleLevel( sPhiPoint, v.vUV, 0, int2( 1, 0 ) );

   // Fetch potential values. Fetches out of domain = 10000
   float4 vCostN = tCostMap.SampleLevel( sCostPoint, v.vUV, 0, int2( 0,-1 );
   float4 vCostS = tCostMap.SampleLevel( sCostPoint, v.vUV, 0, int2( 0, 1 );
   float4 vCostW = tCostMap.SampleLevel( sCostPoint, v.vUV, 0, int2(-1, 0 );
   float4 vCostE = tCostMap.SampleLevel( sCostPoint, v.vUV, 0, int2( 1, 0 );

   float4 vPhi;

   [unroll]
   for( int i = 0; i < 4; i++ )
   {
      vPhi[i] = EvaluateFiniteDifference( vCurPhi[i], vCurCost[i],
                float4( vPhiN[i], vPhiS[i], vPhiW[i], vPhiE[i] ),
                float4( vCostN[i], vCostS[i], vCostW[i], vCostE[i] ) );
   }

   return vPhi;
}
```

***Listing 11 (cont.)*** *HLSL code for iterative eikonal solver*

# Chapter 4

# Using Wavelets with Current and Future Hardware

Mike Boulton[8]
Rare/MGS

**Figure 1.** *Example of using wavelets to compress images*

[8] mboulton@microsoft.com

## 4.1 Introduction

Much of the data we wish to encode over a surface (such as lighting data) is not homogeneous in complexity[9], and is becoming less homogeneous as we pursue higher graphical fidelity.

This data is typically stored using fixed compression methods available in hardware (such as DXT, for example). The main advantages with this approach are that decompression can be performed quickly by the hardware, and also that the size of the data, when compressed, is predictable.

However, this approach also represents a growing inefficiency, since areas requiring a high density of data are under-represented, and vice-versa. In addition, current fixed-compression formats are generally limited to 8 bits per channel.

For applications such as lighting, the problem can be addressed to a certain extent by modifying the injective function which maps the data onto the surface (typically referred to as the *UV Mapping Function*) to map a greater density of texels onto areas of high data complexity, and to also map texels more sparsely over areas of low complexity. Traditionally, this is achieved by either warping the mapping function, or splitting the data up into discrete complexity bands, and uniformly mapping across those bands at a pre-set band density.[10]

There are problems with this approach, however – in particular, when warping the mapping function it is often hard to avoid unpleasant mapping characteristics on the scale of an individual triangle.

Ideally, one would like a variable compression scheme which can focus more on the areas of importance, and less on the areas where not much is happening. This is where wavelets can help!

## 4.2 An Introduction to Wavelets

Wavelets are mathematical functions formed from scaled and translated copies of a single waveform (referred to as the *mother wavelet*). They allow a function to be decomposed into a superposition of different frequency components, which can be operated on individually (this is referred to as *multi-resolution analysis*).

[9] In other words, some areas require us to store considerably more data than others for consistent quality.
[10] See [Hu08] for example.

A function can be put into wavelet form by the use of a *Wavelet Transform*, and can be translated back into the original function via the inverse transform (analogous to a Fourier transform).

Wavelets as basis functions have several significant advantages over the standard Fourier representation (and its analogue on the sphere, spherical harmonics). They are much better at representing functions with discontinuities or sharp changes, functions that are non-periodic, and in many cases have local support, which can permit efficient windowed modifications of the data set. They are a system of hierarchical refinement, and as such can sparsely represent localized areas of low contrast within the data.[11] At the same time, they can be orthogonal.

A wavelet transform can be either continuous or discrete, and can represent data of any dimensionality. In general, we will be interested in discrete two-dimensional wavelets, in particular *2D non-standard Haar*.[12]

See [SDS95] for a simple introduction to wavelets, and how the non-standard 2D Haar basis is constructed.[13] For a more advanced discourse, refer to [DAUBECHIES92].

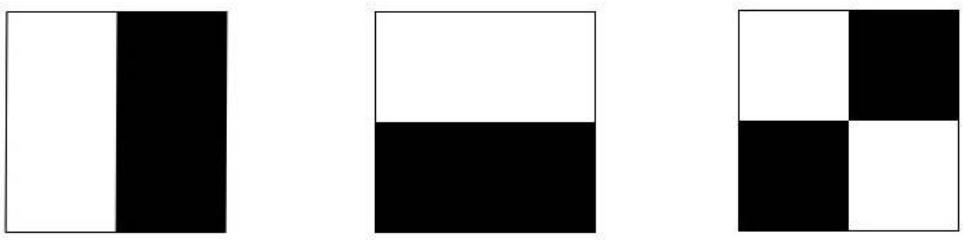

***Figure 2.*** *The vertical, horizontal and diagonal wavelets of non-standard 2D Haar.*

Figure 2 shows a depiction of the three different wavelet basis types for non-standard 2D Haar. The white areas represent +1, and the black areas -1. The scaling function is simply a value of +1 defined over the whole domain.

[11] Since in areas of low contrast, there is very little that actually requires refinement. This is what facilitates compression.

[12] Haar is the simplest wavelet set, and can exhibit blocky artefacts when higher compression is required. This is in general less of a problem for operations such as integration over a hemisphere, except in the case where you have a BRDF whose specular cover is comparable to the cover of the finest wavelet.

[13] Note that we prefer the *non-standard* 2D Haar basis over the *standard* basis because each non-standard basis function has square support which in general produces a sparser representation for the kind of data we wish to compress.

As an example to aid intuition, consider a grid of data with resolution $2^N \times 2^N$, where each cell within the block contains a single real number. We represent the data with this wavelet basis as a tree (commonly referred to as a *wavelet tree*), which in this case has a structure very similar to a quad-tree.

How many wavelets does this grid require in order to be losslessly represented? There is one of each wavelet from figure 2 at the root level, covering the whole grid. These three wavelets tell you how to *refine* the scaling function down to a 2 × 2 block (where each element has dimensions $2^{N-1} \times 2^{N-1}$). The coefficients are stored in the root via a real triple $\{c_v, c_h, c_d\}$.[14] At this stage, each element of the newly-generated 2 × 2 block contains the average of all cells contained within it.

The root has four children, one for each $2^{N-1} \times 2^{N-1}$ block. These children each have their own real triple $\{c_v, c_h, c_d\}$, which is used to refine each $2^{N-1} \times 2^{N-1}$ block down to four blocks each of resolution $2^{N-2} \times 2^{N-2}$ (see figure 3).

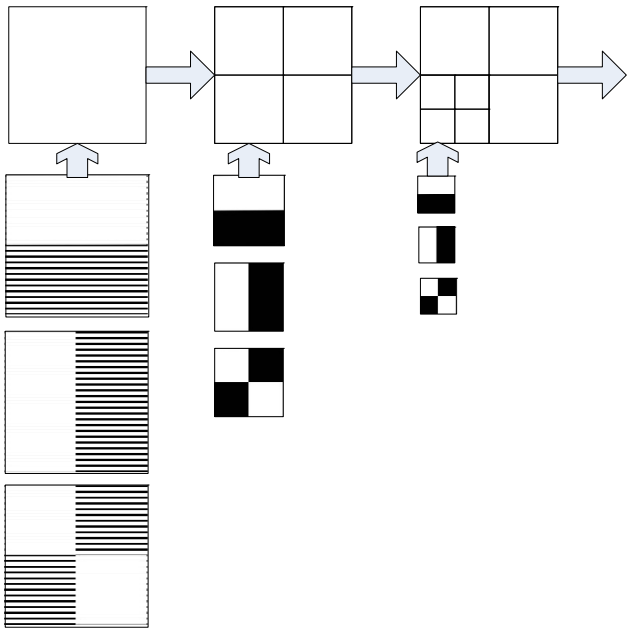

***Figure 3.*** *The first few levels of subdivision.*

This is continued recursively until the whole grid is represented on the level of an individual cell. Note that this is conceptually very similar to traversing a mip-map pyramid, where each level has been generated using a box filter on the preceding level.

[14] Following the convention in [DAUBECHIES92] – $c_v$ is the coefficient for the vertical wavelet, $c_h$ the horizontal, and $c_d$ the diagonal.

So the number of nodes (starting from the root) is $1 + 4 + 16 + \ldots + 2^{N-1} \times 2^{N-1}$, and each node contains three wavelets. Adding this up, and including the single value used to modulate the scaling function, you have exactly $2^N \times 2^N$ terms, which are used to scale $2^N \times 2^N - 1$ wavelets, and a single scaling function.

See listing 1 for pseudo-code to calculate the triple $\{c_v, c_h, c_d\}$ for a node, and listing 2 for pseudo-code to retrieve the refinement value for a supplied coordinate.

```
void CalculateNodeCoefficients( int xMin, int yMin, int xMax, int yMax )
{
   int  xHalf = (xMin + xMax)>>1;
   int  yHalf = (yMin + yMax)>>1;

   // calculate grid average over node cover
   float       nodeAv = GetAverageWithinGridRegion( xMin, yMin, xMax, yMax );

   // calculate grid average over cover of each child
   //  note "UL" stands for "Upper Left", etc.
   float       quadrantAvUL = GetAverageWithinGridRegion( xMin, yMin, xHalf, yHalf );
   float       quadrantAvUR = GetAverageWithinGridRegion( xHalf, yMin, xMax, yHalf );
   float       quadrantAvLL = GetAverageWithinGridRegion( xMin, yHalf, xHalf, yMax );
   float       quadrantAvLR = GetAverageWithinGridRegion( xHalf, yHalf, xMax, yMax );

   // calculate wavelet coefficients for this node
   float       cv = nodeAv - 0.5f*(quadrantAvUL + quadrantAvLL);
   float       ch = nodeAv - 0.5f*(quadrantAvUL + quadrantAvUR);
   float       cd = nodeAv - 0.5f*(quadrantAvUL + quadrantAvLR);
}
```

***Listing 1.*** *Pseudo-code to generate the wavelet coefficients for a node.*

```
// Get refinement for an (x, y) within a particular node with bounds
//  xMin, yMin to xMax, yMax and wavelet coefficients cv, ch and cd.
// Note that this assumes that (x, y) is contained within the bounds.
void GetNodeRefinement( int x, int y )
{
   float       refinement = 0.0f;

   int  xHalf = (xMin + xMax)>>1;
   int  yHalf = (yMin + yMax)>>1;

   // vertical contribution
   refinement += (x < xHalf) ? -cv : cv;

   // horizontal contribution
   refinement += (y < yHalf) ? -ch : ch;

   // diagonal contribution
   refinement += ((x < xHalf) ^ (y < yHalf)) ? cd : -cd;
}
```

***Listing 2.*** *Pseudo-code to calculate a refinement value given an enclosed (x, y).*

Note that the refinement value needs to be scaled according to the area of the corresponding wavelet. So if the root has scale 1, the children of the root have scale ¼ since they cover a quarter of the area (similarly, the grandchildren of the root would have scale one sixteenth).

Compression occurs at this stage by applying non-linear optimization to the wavelet terms. There are several ways to do this – the simplest being to discard any wavelet with a coefficient whose magnitude is below some threshold (referred to as *unweighted selection*). This can be shown to minimize the squared error, but that is not always what you want to do.[15]

## 4.3 Wavelet Applications

Wavelets have many potential applications for real-time rendering. Below is a (far from exhaustive) list of potential applications.

- **Real-time shader texture decompression.** A texture representing arbitrary scalar data (an image or spherical harmonic coefficients for instance) can be compressed lossily into a wavelet tree, which is then compactly represented using a line texture. Given a (*u*, *v*) in the pixel or vertex shader, the value of the original texture at that location can be recovered in real-time using unrolled traversal within the shader.[16] In addition, arbitrary filter operations can be performed hierarchically between the image and a filter kernel (note this includes the bilinear filter kernel). This will be discussed in greater detail in section 4.4.

- **Real-time double and triple product integration for lighting.** Following from [NRH03] and [NRH04], wavelets can be used to encode and compress the elements of the lighting integral. Choosing a wavelet set that is orthogonal (such as non-standard 2D Haar) allows a double-product integral to be decomposed into a sparse list of dot product operations. As described in [NRH04], we can extend this to triple product integration, by describing how to derive the tripling coefficients via a simple (and small) set of rules. In section 4.5 we will discuss how to implement these operations on current hardware, and also discuss an approximation which allows the BRDF to be represented analytically, and its contribution to the integral to be approximated in real-time.

- **Static shadow maps.** Shadow maps that are either entirely static, or mainly static, can be represented in wavelets, potentially with a high degree of compression. Depth-values are queried in the same way as the texture compression described above. Occasional, local changes to the shadow map could be incorporated by only considering those wavelets whose cover intersects

[15] See [NRH03] for a discussion of three methods for non-linear lighting approximation.

[16] As will be discussed in part 4.4, a $1024^2$ grey-scale image with sub-blocks of size $16^2$ can be decompressed on the Xbox 360™ at a resolution of 1280×720 at over 500Hz.

the area of change. This demonstrates the value of having a basis set with local support.

- **Displacement map compression.** Similarly, displacement maps can be wavelet compressed. This can be used for e.g. morph target (or *blend-shape*) compression – in the case of many morph targets applied to a single mesh, typically only a modest subset of the vertices are significantly displaced. Wavelet compression would heavily compress areas of low change, and represent areas of high change to an arbitrary level of precision. Additionally, wavelet compression is a useful method for compressing very large displacement maps for e.g. terrain representation – here (similar to morph targets) you often have smallish areas of high frequency change scattered around, with smooth low-frequency data in-between. With reasonable compression ratios, this would allow a single texture (here storing a wavelet tree rather than the displacement values exactly) to span areas much larger than the maximum texture resolution that current commodity graphics hardware would allow, and might allow for easier streaming and LOD methods.

- **Easier static and dynamic texture packing.** Automatic generation of a UV mapping function over an arbitrary mesh is a tricky problem, made more so when additional mapping characteristics are required, such as distortion minimization, and trying to match texels up across atlas boundaries.[17] In addition to these requirements, we also want to maximize total texture usage in order to efficiently use available memory. There are several programs available which can generate good UV mappings,[18] but in general there are still cases where mappings with poor texture usage are generated. Wavelet image compression will heavily compress the unused gaps between polygons, and can produce good results even for near-lossless compression.[19] For dynamic packing, one could adopt a scheme where new texture space was allocated in large blocks with no effort to effectively tile with existing atlases. When filled, this block would be wavelet compressed.

- **Geometry representations.** A deformable object with associated UV mapping can have the deformations represented by wavelets over the surface. One possible approach would be to allow a fixed upper limit on the number of wavelets to be used (to control memory usage, perhaps). Space could be made for new deformations by removing the oldest existing high-frequency wavelets – so the broad shape of older deformations could be maintained for a longer time, at the expense of the finer details. Additionally, the multi-resolution

[17] See [HLS07] for a good reference on mesh parameterization.
[18] *UVAtlas* in the DirectX® SDK June 2008 for instance.
[19] Because the unused gaps will generate lots of zero-valued wavelet coefficients, and they can be freely pruned without compromising reconstruction fidelity.

representation could be used to directly update the centre of mass and moment of inertia of the object, potentially at a different level of accuracy. This suggests that data represented in a single multi-resolution form is easier to use across different systems with varying precision requirements.

## 4.4 Real-Time GPU Image Decompression

In this section, a method for real-time wavelet decompression of an 8-bit monochrome texture by a vertex or pixel shader is described.[20] Note this can easily be extended to color textures by storing each channel as a separate wavelet tree,[21] and also can be extended to handle any data type (float for example).

In section 4.2, the generation of a wavelet tree from a block of data was discussed. Given a texture, we firstly partition it into fixed-size sub-blocks. For each sub-block, we generate a wavelet tree. This tree is pruned recursively as follows:

1. Flag every node which is a leaf and where the absolute value of each of the three wavelet coefficients falls below a user-defined threshold.
2. For every node which is not a leaf, and which has all four children flagged, prune all four children.
3. Repeat procedure until no further pruning can be performed.

Note that a node is only pruned if all three associated siblings are also pruned. This is a concession to allow easier tree indexing on the GPU, and does not add a significant overhead to total storage requirements. The scheme can be extended to support more fine-grain pruning at the cost of a more complicated traversal process.

The method used to prune a tree can be entirely user-defined, and can easily be extended to incorporate an *importance mask* – for example, there could be areas of the data which, although lower in contrast, the user still wishes to preserve at a greater fidelity, perhaps losslessly (and vice-versa). The importance mask value can be used to modify the pruning function on a per-texel level.

Each tree is stored linearly and breadth-first, with each node packed into a single ARGB8 texel,[22] where {*r*, *g*, *b*} stores the quantized and windowed wavelet coefficients {$c_v$, $c_h$, $c_d$} and {*a*} stores the linear offset to the first child of this node, if a child exists.

[20] See [DCH05] for a description of a similar approach.

[21] Although you might consider an HSV (or YCbCr) encoding rather than RGB, and store *hue* and *saturation* at a coarser level than *value*. See http://en.wikipedia.org/wiki/Jpeg2000.

[22] The actual texture format used is `D3DFMT_LIN_Q8W8V8U8` since this automatically maps the wavelet coefficients onto [-1,1] which saves a few shader instructions.

All of the trees are then consecutively packed into a single line texture.

We choose to firstly partition the image into sub-blocks (in this case, sub-blocks of resolution 16 × 16 texels) for two reasons. Firstly, it allows each sub-block wavelet tree to lie within at most two texture cache lines,[23] and secondly it allows a single unsigned 8-bit integer to be used per node to encode the tree structure.

In addition, we require a texture at the resolution of one sub-block per texel to encode the scaling function coefficient, and record the offset from the start of the line texture where the associated sub-block wavelet tree begins. See Figure 4 for more details.

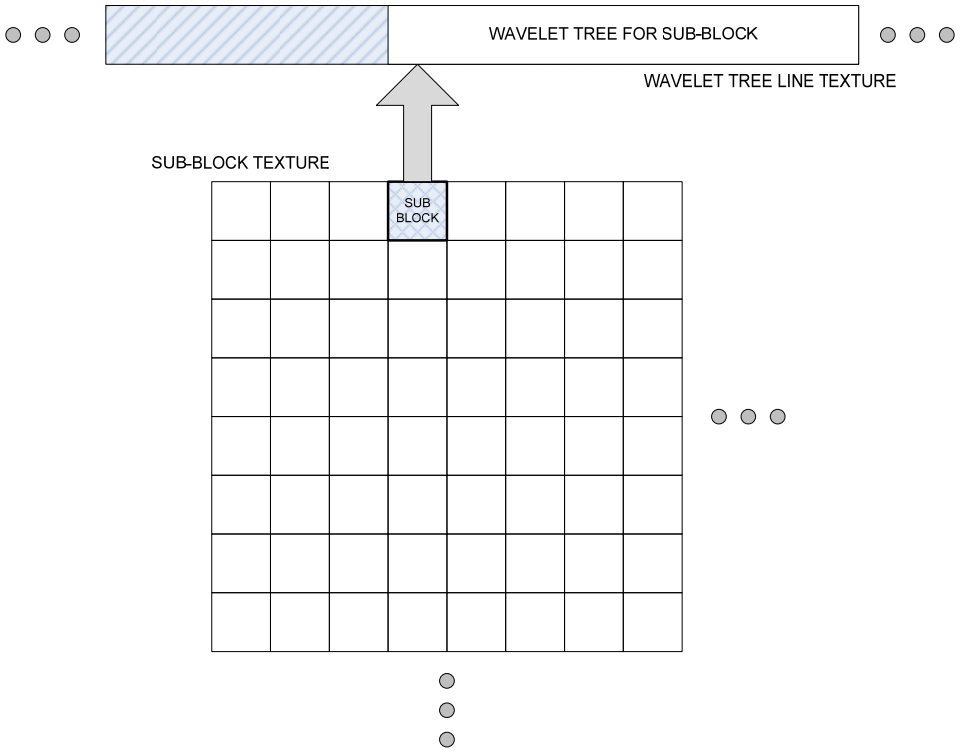


***Figure 4.*** *Each texel in the sub-block texture contains an offset to the wavelet tree.*

So given a (*u*, *v*) within the unit square, we reconstruct the value as follows:

1. Fetch the sub-block texel containing the given (*u*, *v*).
2. Use the wavelet tree offset to point to the start of the sub-block wavelet tree in the wavelet tree line texture.

[23] This is important for performance since we are traversing a tree on the GPU, which in general is not a good fit for fetch coherency and texture cache usage.

3. Perform depth-first traversal down to the leaf containing the given (*u*, *v*) accumulating the contribution made by each wavelet node along the way. As soon as traversal completes, attempt to jump to the end of the shader.[24]
4. Add the result to the scaling term contained within the sub-block texel.

See Listing A1 in the appendix for an optimized Microsoft Xbox 360™ microcode shader which performs these steps.

Figure 5 shows the results for three different cases on a monochrome $1024^2$ texture.

1. Reference case. Here we have a standard uncompressed texture rendering via a single texture fetch. Memory size (with mip-maps) is 2MB.
2. Wavelet compressed with cut-off value of 0.05. The total memory requirement for the wavelet tree line texture, and sub-block texture is 249KB (representing around an 8:1 compression ratio). It is running full-screen at 1280 × 720 with a speed of 579fps.[25]
3. Wavelet compressed with cut-off value of 0.082. The total memory requirement is 157KB (13:1), and it runs with a speed of 622fps.

Notice that as the cut-off is increased, the frame rate also increases. This is because the shader attempts to perform a dynamic branch to the end of the program whenever it encounters a leaf prematurely. For higher compression, more premature leaves will be encountered and so this will yield a greater benefit.[26]

Because of the refinement-based nature of wavelets, a separate mip-map chain is not required, since by simply terminating traversal prematurely the box-filtered mip-map value at that level is obtained.

See figure 6 for a zoomed-in comparison. Note that at higher compression ratios, the blocky artifacts discussed previously are evident in areas of lower contrast. Notice however that even at these high compression ratios, fine high-contrast detail is still represented to a high degree of accuracy (note particularly the collar and buttons).

[24] We would ideally also want to terminate shader traversal if screen-space minification was encountered.
[25] Note that this frame rate also includes a small amount of unrelated system overhead, such as clearing and resolving screen buffers, etc.
[26] On the Xbox 360™, the GPU operates on vectors of 64 pixels wide. For a dynamic branch to be effectively executed (and the performance benefits gained), every element of the vector must follow the same branch path.


***Figure 5.*** *Comparison between reference, cut-off 0.05 and cut-off 0.082.*

***Figure 6.*** *Zoomed-in comparison for all three cases.*

An associated user-generated importance mask could improve the overall quality of the most compressed case, whilst maintaining the same compression ratio. In particular, if areas around the arms and face of both figures were increased in relative importance a more agreeable result might be obtained.
However, for general images this technique does not often produce results significantly better than DXT5A, for example (which has a fixed compression ratio of 8:1), and for many image applications the blocky artifacts produced by the 2D Haar basis would not be acceptable. Image compression does serve as a good method for demonstrating wavelet compression intuitively, but it is argued that the practical application of real-time wavelet texture decompression is to textures storing general data, such as lighting, which is not of such a homogeneous nature.

For better wavelet compression of general images, more sophisticated wavelets are typically used, for example the *Cohen-Daubechies-Feauveau* wavelet (which is part of the JPEG2000 standard[27]). However, due to the complexity of this wavelet, real-time decompression at interactive rates is not really feasible on current hardware.

Hierarchical filtering between multiple wavelet-compressed images can be efficiently performed. In addition, filtering between an image and an analytically-defined kernel can be performed (the bilinear filter kernel, for instance[28]). The only caveat here is the case where a kernel overlaps a sub-block boundary. This can be addressed by splitting the process up into multiple passes, based on the size of the smallest kernel, or alternatively by overlapping the sub-blocks slightly at a slight cost to overall compression.[29] Here, it is often better to encode the tree in a depth-first manner, rather than breadth-first.

[27] See http://en.wikipedia.org/wiki/Jpeg2000 for example.
[28] There are at least two different ways to approach bilinear filtering – one would be to perform four individual fetches using the shader in Listing A1, and apply the bilinear weights. The other would be to perform a genuine pruned depth-first tree traversal between the image and a per-pixel defined bilinear filter kernel. See section 4.5.
[29] Note this becomes less feasible the larger the filter kernel. The bilinear kernel would only require an overlap border of width one, however.

This is essentially what is being done when calculating the double-product integral for relighting – here, we are performing a filter between the image over the sphere representing the incoming light, and the image over the sphere representing the local visibility and diffuse BRDF (this is discussed in more detail in the next section).

See figure 7 for an example of wavelet-compressed textures containing spherical harmonic coefficients of order 0 and 1 (with resolution $1024^2$). In this case, DC and linear spherical harmonic coefficients were stored as 16-bit floats, and wavelet compressed.

| **CUT-OFF** | Overall pruning | DC pruning | Linear pruning |
|---|---|---|---|
| 0.00008 | 94.0% | 94.2% | 93.9% |
| 0.00016 | 95.2% | 95.5% | 95.1% |
| 0.0004 | 97.9% | 98.0% | 97.8% |

***Table 1.*** *Pruning results for wavelet compression of spherical harmonic data.*

Table 1 shows the percentage of total nodes in the wavelet tree pruned (using the pruning method previously described) as a function of user-defined cut-off.

Here wavelets are able to provide a very high compression rate, whilst still maintaining good reconstruction quality around areas important for perceived lighting fidelity (in particular, note the soft area shadow around the base of the cuboids). Additionally, the pruning approach used here is rather clumsy, and a better algorithm would likely allow further compression.

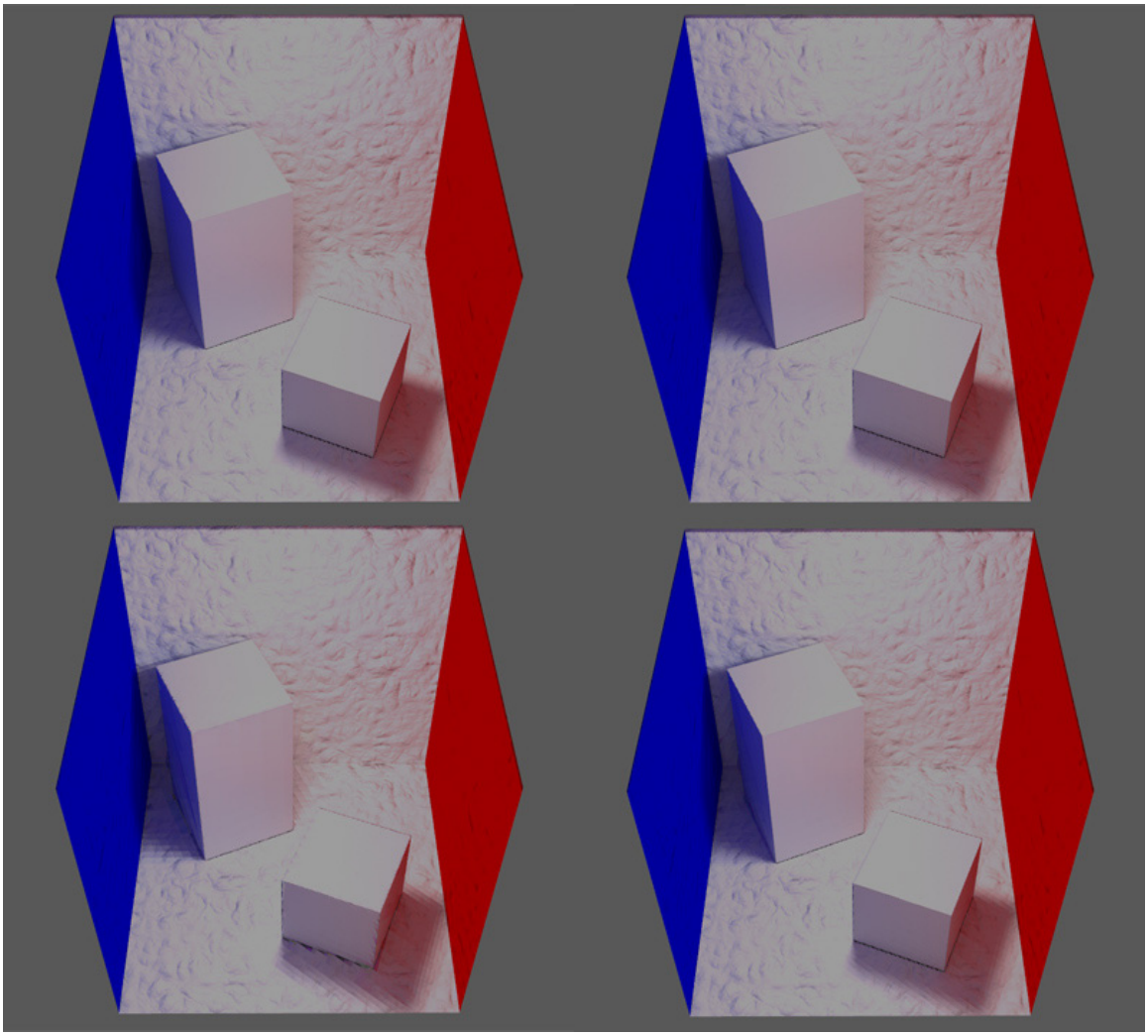

***Figure 7.*** *Spherical harmonic lighting with wavelet-compressed coefficients. Clockwise from top-left: Reference, cut-off 0.00008, cut-off 0.00016, cut-off 0.0004.*

With spherical harmonic lighting on current hardware, one often needs to make a difficult decision about whether to include quadratic SH terms, or just use linear, since it is often the case that quadratic terms only make a noticeable difference to diffuse lighting in certain places, although they require five more coefficients to be stored per channel. A wavelet approach would help here, since the large portions of low-contrast data stored in texture channels representing the higher bands would be automatically compressed away.

Since linear spherical harmonics require twelve channels to represent a color signal, one might think that performing twelve sets of wavelet decompression in the shader would be unrealistic. However, with such high compression ratios, the typical decompression traversal would not likely descend many levels from the root.[30] In addition, representing the light in HSV format for instance would allow two of the trees representing the color information to be coarser still.

## 4.5 Lighting

Orthogonal wavelets can be applied to the rendering equation in a very similar way to spherical harmonics. The framework for performing double-product integration with 2D Haar wavelets was introduced in 2003 by [NRH03], and extended to triple-product integration a year later [NRH04].

Why use wavelets for lighting instead of spherical harmonics? Spherical harmonics struggle to efficiently encode spatially-compact areas of high variability. As with the Fourier series, they exhibit unpleasant ringing artifacts when trying to resolve sharp discontinuities.[31]

In general, a wavelet approach to lighting requires 2 orders of magnitude fewer coefficients [NRH03], and makes the accurate calculation of high-frequency environmental specular responses much more feasible at rates approaching real-time.

Additionally, wavelet bases with local support can in theory support staged, windowed modifications to transfer functions. For instance, consider the situation where a wall has had a hole punched in it (via a Boolean operation for instance). The hole would allow light through, which would make many surrounding transfer functions within the lightmap invalid. Although rebuilding those transfer functions in real-time might be very costly, the multi-resolution nature of wavelets can allow this cost to be prioritized, and the work spread over many frames. For instance, the highest priority work would likely be updating those transfer functions which receive direct lighting through the hole.

[30] And since one typically encounters large, coherent patches of low frequency lighting, dynamic predication is likely to work well.

[31] This can be addressed to a certain extent at the cost of feature blurring. See [SLOAN08].

Further, one would like to focus on the low-frequency wavelets first, in order to quickly carve out the approximate shape of the cast light. So only the low-frequency wavelets whose cover intersects the volume that the hole subtends onto the sphere would be initially considered.

The shape of the hole borders would then be refined, in addition to updating transfer functions with no direct view of the hole, but which need to change in order to compensate for additional bounce lighting.

This would allow worlds that are not entirely static to benefit from a full GI solution, provided that geometric changes could be controlled so as not to saturate the hardware.

For fast-moving objects, the shadowing produced would have no latency, and a fidelity that is related to surrounding lighting complexity and power of the hardware.

This would be a tricky thing to do with spherical harmonics, since the bases have global support, and could not easily be prioritized in the above fashion.

However, two significant advantages that spherical harmonics do possess (that wavelets do not) is rotational invariance and easy analytic rotation of the basis functions (at least for low-order SH). This poses a significant problem when evaluating the rendering equation for wavelet-compressed components.

For the case of the Phong BRDF[32] for instance, one would like to rotate the diffuse component of the BRDF (a cosine lobe) so that it is oriented along the normal, which might be obtained from a high-frequency normal map which has many texels overlaying a single lightmap texel. Similarly, one would like to rotate the specular component (a cosine lobe raised to a power) so that it is oriented along the reflected light direction.

In [NRH04] we see that this can be addressed by re-parameterizing the BRDF about the reflected light direction, and then generating a set of samples over all remaining variables. This dataset is then wavelet compressed using aggressive non-linear optimization.[33] When performing rendering, the closest appropriate BRDF record is selected from this compressed dataset.

We can also choose to encode the BRDF and visibility term in the local frame of the lightmap texel (as described in [MHL*06]), and rotate the lighting environment into this

[32] Here the cosine term is included in the BRDF.

[33] [NRH04] uses sampling of (6×64×64)×(6×128×128) for $\omega_r \times \omega$ for Phong (where $\omega_r$ is the reflection vector), which can handle specular powers up to 200. For 99% accuracy, only 0.1-1% of the terms are required.

local frame.[34] The lighting environment is compressed at a sampling of orientations in a similar manner to BRDF compression mentioned above. The calculation is then performed in local space, instead of world space. This is a smart idea because the lighting function is 2D, but the BRDF is 4D, so storage requirements for a compressed sampling of lighting functions will generally be significantly less than the storage requirements for a BRDF representation. However, since the visibility is locked into the same frame as the BRDF, it can't be rotated to correctly compensate for BRDF orientation around high frequency normal map directions, which leads to incorrect solutions.[35]

We will later argue that a relatively simple isotropic BRDF (such as Phong) can be treated as a separate entity, whose contribution to the integral can be naturally folded into the traversal process, and approximated via importance sampling.

See [NRH03] for a good direct comparison between spherical harmonics and wavelets for lighting.[36]

$$B(x, \omega_o) = \iint_{\Omega} L(x, \omega_i) V(x, \omega_i) \rho(x, \omega_i, \omega_o)(\omega \cdot n) d\omega_i$$

***Equation 1.*** *The rendering equation for direct illumination.*

See equation 1 for the rendering equation. Note that it is common practice to incorporate the cosine term $\omega \cdot n$ into the BRDF $\rho$. The function *L* represents the incoming lighting, *V* the local binary visibility, $x$ is the point on the surface, $n$ is the normal, and $\omega_i$, $\omega_o$ represent the incident and outgoing lighting direction respectively.

If the integrand is rearranged into the form *AB*, where *A* is to be varied with respect to *B*, the integration is referred to as *double product*. Similarly, if arranged into the form *ABC* where all three parameters vary with respect to each other, the integration is referred to as *triple product*.

Since non-standard 2D Haar forms an orthogonal basis set, double-product integration decomposes into a sparse set of dot product operations in exactly the same way as with spherical harmonics, since all off-diagonal terms vanish (see equation 2).

[34] [MHL*06] uses s*pherical wavelets* (see [SCHRÖDERSWELDENS95]), with a basis set that is isomorphic to 2D Haar.

[35] Since in order to get a bump-mapped response, the lighting environment is being rotated in the opposite direction so that it has the correct orientation with respect to the fixed BRDF for each normal from the normal map. You need to do this to the visibility function as well, but it is locked into the same frame as the BRDF.

[36] In particular, see figures 2 and 3 in [NRH03].

$$\iint_{\Omega} A(\omega)B(\omega)\,d\omega = \iint_{\Omega} \left(\sum_i a_i \Psi_i(\omega)\right)\left(\sum_j b_j \Psi_j(\omega)\right) d\omega$$
$$= \sum_i \sum_j a_i b_j \iint_{\Omega} \Psi_i(\omega)\Psi_j(\omega) d\omega = \sum_i \sum_j \delta_{ij} a_i b_j = A \cdot B$$

***Equation 2.*** *Decomposition of double-product integration into a sparse dot product.*

Unfortunately, triple-product integration doesn't turn out to be quite this simple, since in general the integration between three orthogonal basis functions does not decompose to something as nice as the Kronecker delta, but a *tripling coefficient* instead. Although in general these can be unpleasant to calculate,[37] [NRH04] shows that for 2D non-standard Haar they can be obtained via the application of a simple and small set of rules.

We now consider implementing a few different approaches on hardware. Firstly, consider the case where we would like to vary the lighting *L* with respect to the rest of the integrand, and for simplicity assume that the BRDF is diffuse-only and that normals are taken from the vertices, rather than a normal map.

A UV mapping function is generated for the lightmap. For every texel within the lightmap, visibility information is generated over the sphere centered at the texel. Each record on the sphere containing binary visibility information is then weighted by the clamped inner product with the interpolated vertex normal.

This data is then projected onto a discrete decomposition of the sphere, such as a cube map, and an appropriate correction term is applied if required,[38] and then transformed into a wavelet representation. Non-linear optimization is then applied.

The above process is typically performed using a ray tracer (or photon mapper), but in this case can be performed very quickly to a reasonable approximation using rasterization into a cube map on the GPU.[39]

Incoming environmental lighting (assumed to be effectively from infinity) is then sampled onto the same sphere decomposition, for an appropriate set of values and/or orientations. In this example, we generate environment lighting sets for *Grace Cathedral* and also for a home-made red area light. Both environments are rotated one

[37] For spherical harmonics, these tripling coefficients are referred to as *Clebsch-Gordan coefficients* and are rather complicated.

[38] The cube map requires such a correction, since texels close to the corners subtend a smaller solid angle than texels close to the center of a face. See [FujitaKanai04].

[39] Although if bounced lighting etc. is required, this isn't really feasible. Additionally, if the spherical decomposition is not a cube map itself, the data will need to be re-sampled appropriately.

full revolution around the *X* axis, with 256 equal angular subdivisions. Each frame is compressed in the same way as the per-texel data.

Notice that the per-texel data and the environment lighting data are both defined in world space. This allows direct integration without the need to rotate. It is very important that both the environmental lighting and per-texel transfer functions are in the same space, and use the same decomposition, since we require an exact correspondence between associated wavelet bases.

When performing rendering, we firstly select the most appropriate record from the environmental lighting data set, based upon current orientation. This record contains three pruned wavelet trees, one for each color channel. Each tree is to be integrated against the transfer function wavelet tree for each texel in the lightmap.

As discussed above, the per-texel double product integration that needs to be performed decomposes to just a sum of dot product operations, where the dot product is performed between the triple $\{c_v, c_h, c_d\}$ from each node in the first tree with the triple $\{c'_v, c'_h, c'_d\}$ from the corresponding node in the second tree, if it exists.

Let *A* be the set of linearized indices of all nodes in the first tree. Let $\bar{C}_A^i$ be the triple $\{c_v, c_h, c_d\}$ at node $i$ (appropriately area-weighted), and $S_A$ be the scaling function coefficient of the first tree. Similarly for the second tree (replacing *A* with *B*).

$$I = S_A S_B + \sum_{\forall\, i \in A \cap B} \bar{C}_A^i \cdot \bar{C}_B^i$$

***Equation 3.*** *Calculation of per-channel intensity.*

See equation 3 for the calculation of per-channel intensity. Note that only the intersection of both trees needs to be traversed for evaluation, since if any node in one tree does not have a corresponding node in the other, there will be no contribution.[40]

So to perform this operation in a shader for instance, both trees need to be traversed in parallel. Each node pair has their contributions added to the total (as in equation 3). However, if the node currently being considered from one tree has children but the corresponding node from the other tree does not, the evaluation function needs to be able to jump over all children of this node (and their descendants) to the next sibling or ancestor (which due to the pruning process described previously will always correspond to the node from the other tree). Traversal is then continued as before until the root is encountered.

[40] Note that this doesn't *directly* follow from equation 3, since there could be a case where a parent was pruned which had children that were not. However, our pruning process described earlier prevents this from happening (and it's actually fairly unlikely anyway).

We modify the process used in section 4 for image decompression as follows: the tree is now depth-first, and stored into a line-texture as before. The alpha channel is now used to store the tree level of the node immediately following the current node in the line texture. If the next level is greater (in other words, the next node is a child of the current node), a special *jump node* is stored in the next texel, between the current node and the child. This node contains the linear offset (in texels) from the current node to the next node on a level less than or equal to it (a sibling, parent or other ancestor).

If during traversal a disagreement is encountered concerning the next node, the jump node is used to entirely omit the descendant branch of the offending tree.

See Listing A2 in the appendix for a microcode shader which performs these steps, and figure 8 for a screenshot of a simple model (a torus over a plane) under both lighting conditions. Here, the UV mapping was generated with a modified version of UVAtlas, and the lightmap has a resolution of $128^2$ texels. Each transfer function and environment lighting function was first rendered into a $32 \times 32 \times 6$ cube map, and then compressed (one wavelet tree per cube map face). The transfer functions across the whole lightmap required 8.7MB of total storage. We recorded an average frame rate of over 250Hz was recorded[41], when using the red area light; and for Grace Cathedral just over 40Hz.

Note that this implementation can be significantly optimized – it is currently bound by texture cache stalls because of both jumping behavior, and also the fact that transfer function wavelet trees of adjacent lightmap texels are stored a long way away from each other. We discuss optimization approaches towards the end of the section, including a method of interleaving trees representing localized data. Additionally, predication behavior is quite poor since jobs of a similar size are not in general clustered together.

[41] Again, running on the Xbox 360™.

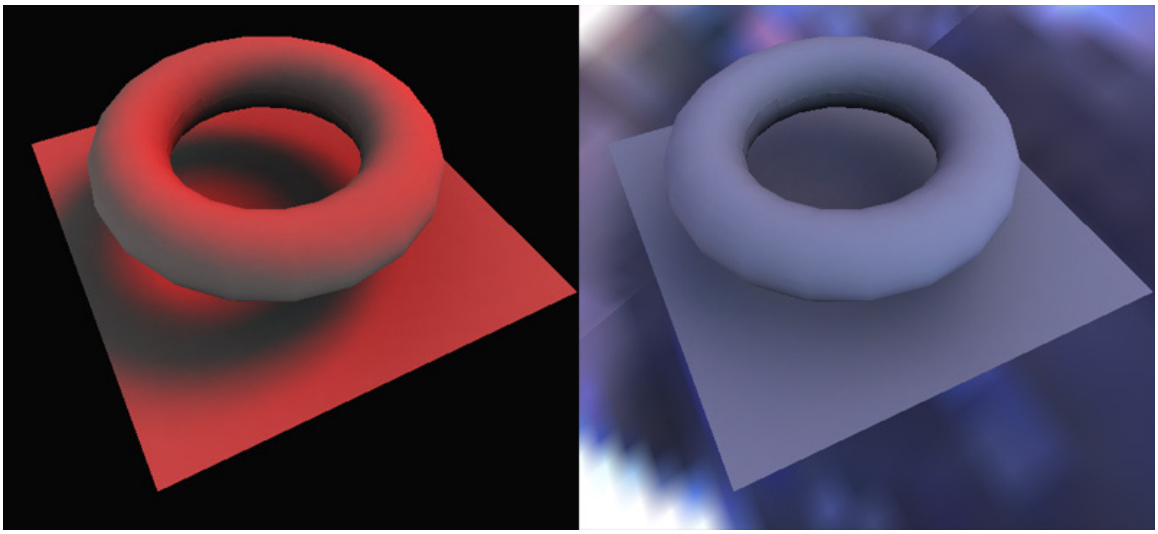

***Figure 8.*** *Real-time GPU double product integration between (left) a red area-light and (right) Grace Cathedral lighting environment.*

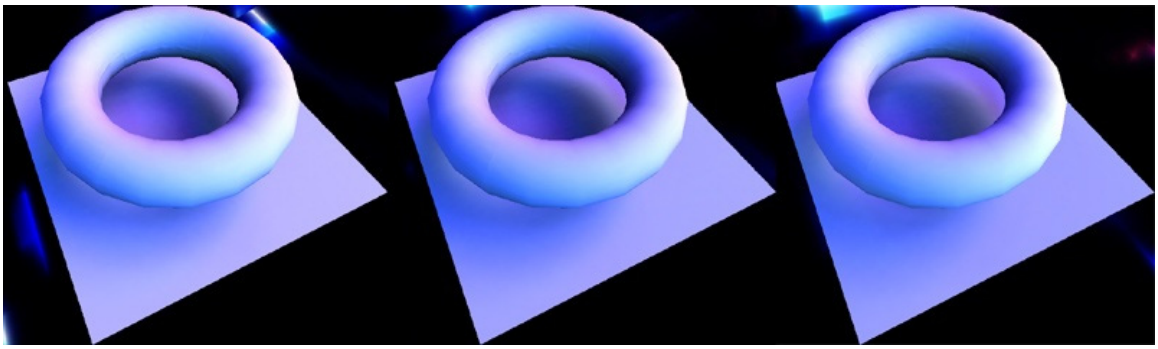

***Figure 9.*** *Three consecutive frames of Grace Cathedral as above, but with exaggerated contrast.*

Notice that even though not entirely optimal, there is a large performance difference between grace cathedral and the much simpler red area light. This is because the red area light has an extremely sparse wavelet representation when compressed, so the intersection between this and any arbitrarily complicated transfer function is always guaranteed to be small.

Returning to figure 8, notice the quality of the torus shadow on the left – this is well beyond the resolving power of quadratic or cubic SH. However, under the smoother lighting conditions such as Grace Cathedral represented in low dynamic range, the results become much more comparable to low-order SH for diffuse lighting (which is expected, see [[RAMAMOORTHIHANRAHAN01]]).[42] See figure 9 for three consecutive frames where the contrast has been artificially increased – here, sharper shadow boundaries are clearly visible.

[42] Note that although the Grace Cathedral case looks a bit bland, it is quite convincing in motion. See associated talk.

Next, consider how one might uncouple the BRDF from the visibility function. If instead we fixed incoming light and local visibility, and chose to vary the BRDF using a user-specified high frequency normal map for diffuse (and specular) bump-mapped response, we would have another double-product integral, and could address the difficulty of orienting the BRDF about arbitrary normals and view directions by adopting the approach in [NRH04], i.e. by compressing a sampling of BRDFs.[43]

However, consider the case where we remove the cosine from the double integral, and rather than performing the integration according to equation 3, we instead perform a point evaluation of the lighting and visibility wavelet tree in a similar way to the method of image decompression discussed in the previous section. At each leaf, we multiply each of the four refined values by an approximation of the BRDF over each individual patch, and add the results to the accumulating total.

See Listing 3 for pseudo-code demonstrating this process.

```
→For each lightmap texel

  →For each transfer function wavelet tree

     →Starting from root node

       Refine current quadrant value

       Am I a leaf?

         →Yes: Multiply current refinement value against approximation of BRDF integral
            over current patch, and add this to the total.

         →No: For each child

           Does my cover intersect the BRDF kernel?

             →Yes: Descend to child

             →No: Do not descend to child
```

***Listing 3.*** *Pseudo-code for approximating the integral between an analytically-defined isotropic cosine-power kernel and a wavelet-encoded transfer function.*

Triple product integration between two functions and the BRDF can also be approximated in a similar way using double product integration, and again multiplying the leaf values with the approximation of the BRDF integral over the patch.

How can the estimate be performed? For certain simple BRDFs, and certain spherical decompositions,[44] direct analytic integration may be possible. Even more complicated

[43] However, it might be a challenge to encode this in a cache-friendly way, especially for a noisy normal map.

[44] The cylindrical mapping is a good one here, since each patch is just a spherical polar square. Unfortunately, there is rather extreme distortion.

decompositions such as *HEALPix*[45] can permit relatively straight-forward analytic integration over each patch (although it does get a bit fiddly in the polar regions, see the appendix in [[GHB*05]]). The problem however is complicated by requiring that the BRDF be clamped to zero over the negative hemisphere with respect to the normal direction, and if the analytic integral is available in closed form, it will likely be complicated and expensive to set up and evaluate.

Alternatively, an SRBF approach could be used. Here, we would approximate each rectangular patch with a circle, and perform integration by using the inner product between the patch center and the kernel center to index into a pre-computed integral table.

A third option would be to approximate the integral by taking point samples over the patch, and averaging their values, calculated analytically.

This is a very good fit for HEALPix, which is hierarchical and has a well-defined method for *nested indexing* (see [GHB*05]). This allows easy generation of unbiased points across any HEALPix pixel, and because of the hierarchical arrangement, it would be easy to generate a quad-tree across the patch to better focus each inter-patch sampling (effectively performing importance sampling). Additionally, HEALPix has a host of other very attractive features, such as equal-area patches and low shape distortion that make it ideally suited as a spherical decomposition for wavelet lighting.[46]

To implement this approach on the GPU, one needs to have the concept of a stack to avoid the need to look backwards in the tree to recover lower level values when ascending from a child during traversal. To implement a scalar stack of fixed size 8, for instance, is a simple operation in the shader requiring only two ALU cycles for a push or pop, and 2 GPRs. Here however we require four floats per push/pop per tree, which can only be performed by using a large block of GPRs and a list of *mov* instructions. See figure 10 for three consecutive frames using this approach.

It also seems feasible to implement this on the CELL™ processor, where sections of tree can be streamed into the local store. Fast reads from the local store during traversal are likely to be better than trying to organize traversal to best utilize existing texture caching methods. Additionally, job organization for best branching/predication performance is likely to be easier.

[45] See [GHB*05] and [WANWONG07].

[46] As mentioned previously, *spherical wavelets* have also been used, with similar benefits. Here the basis is triangle-shaped. See [MHL*06] and [SCHRÖDERSWELDENS95].

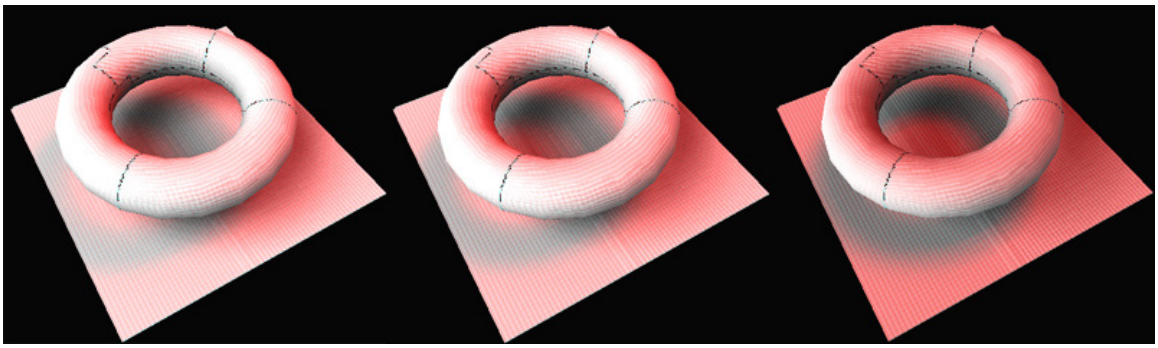

***Figure 10.*** *Three consecutive frames of triple-product integration using point sampling for BRDF integral approximation, and a high-frequency normal map for diffuse lighting.*

## 4.6 Conclusion

A practical introduction to wavelets has been provided, in addition to demonstrations of real-world examples.

The author believes that methods of variable compression, and "shader tree traversal" algorithms like the ones demonstrated, are likely to play an important role on future hardware, to help better focus existing memory and bandwidth resources.

This approach also allows a generalization to occur concerning the methods we use to parameterize data. "Voronoi diagram" texture maps and 2D tangent space BSP trees are an interesting example of how we can move forward to "point cloud" type representations of data, which not only provide a more honest account of the underlying information, but also allow more ambitious methods for both its modifying and handling.

Although this increased generality does seem to come with a certain "price of entry", it has hopefully been demonstrated that modest progress can be made even on current hardware.

## 4.7 Appendix

```
  xps 3 0
  config AutoSerialize=false

  dcl texcoord0 r0.xy

  // for 1024x1024 there are 64x64 subblocks of size 16x16
  def c0, 1.0, 64.0, 0.015625, 0.0078125
  def c1, 0.00390625, 2047.0, 1024.0, 0.5
  def c2, 2.0, -1.0, 127.0, 127.5

  //
  // fetch the grid values
  //           z              = base scale at 16x16 cell
  //           (x, y)         = offset in coefficient texture to start fetching
  //

  tfetch2D r2.yzx , r0.yx, tf1, MagFilter=point, MinFilter=point, FetchValidOnly=false

  // calculate initial quadrant dividers
  mul r1.xy, c0.y, r0.xy
  floor r1.xy, r1.xy
  + movs r1.z, c1.x
  mad r1.xy, r1.xy, c0.z, c0.w

  serialize

  // scale index to required range
  mad r2.xy, r2, c1.y, c1.w
  floor r2.xy, r2
  mad r2.x, r2.y, c1.z, r2.x


////////////////////////////////////////////////////
// to 8x8
////////////////////////////////////////////////////

  // fetch coefficient & index
  tfetch1D r4.xyzw, r2.x, tf0, MagFilter=point, MinFilter=point,
      UnnormalizedTextureCoords=true, FetchValidOnly=false  // from 8:8:8:8

  // find out which quadrant we're in, and build the basis
  sge r3.xy, r0.xy, r1.xy
  sgt r3.zw, r3.xyxy, r3.yxyx
  dp3 r3.w, r3.xyy, r3.xyy
  + adds r3.z, r3.zw
  mad r3.xyz, r3, c2.x, c2.y

  // calculate new half-level coordinates
  mad r1.xy, r3, r1.z, r1.xy

  serialize

  // scale the index to the correct range, and round
  mad r0.z, r4.w, c2.z, c2.w

  // per-level rescaling
  mul r4.xyz, r4, c8.x
  + floors r0.z, r0.z

  // add contribution
  dp3 r4.w, r4, r3
  + sgts r0.w, r0.z
  add r2.z, r2.z, r4.w

  // half size of quarter-level difference
```

```
  + mulsc r1.z, c1.w, r1.z

  add r2.x, r2.x, r0.z

  // Abort?  Will abort if fetch points to (0, 0)
  + setp_eq r0._, r0.z

  // abortion jump
   (p0) jmp L1

  // point to selected node in texture
  mad r2.x, r2.x, r0.w, r3.w

////////////////////////////////////////////////////
// to 4x4
////////////////////////////////////////////////////

  // fetch coefficient & index
  tfetch1D r4.xyzw, r2.x, tf0, MagFilter=point, MinFilter=point,
       UnnormalizedTextureCoords=true, FetchValidOnly=false // from 8:8:8:8

  // find out which quadrant we're in, and build the basis
  sge r3.xy, r0.xy, r1.xy
  sgt r3.zw, r3.xyxy, r3.yxyx
  dp3 r3.w, r3.xyy, r3.xyy
  + adds r3.z, r3.zw
  mad r3.xyz, r3, c2.x, c2.y

  // calculate new half-level coordinates
  mad r1.xy, r3, r1.z, r1.xy

  serialize

  // scale the index to the correct range, and round
  mad r0.z, r4.w, c2.z, c2.w

  // per-level rescaling
  mul r4.xyz, r4, c8.y
  + floors r0.z, r0.z

  // add contribution
  dp3 r4.w, r4, r3
  + sgts r0.w, r0.z
  add r2.z, r2.z, r4.w

  // half size of quarter-level difference
  + mulsc r1.z, c1.w, r1.z

  add r2.x, r2.x, r0.z

  // Abort?  Will abort if fetch points to (0, 0)
  + setp eq r0. , r0.z

  // abortion jump
   (p0) jmp L1

  // point to selected node in texture
       mad r2.x, r2.x, r0.w, r3.w


////////////////////////////////////////////////////
// to 2x2
////////////////////////////////////////////////////

  // fetch coefficient & index
  tfetch1D r4.xyzw, r2.x, tf0, MagFilter=point, MinFilter=point,
      UnnormalizedTextureCoords=true, FetchValidOnly=false // from 8:8:8:8

  // find out which quadrant we're in, and build the basis
  sge r3.xy, r0.xy, r1.xy
  sgt r3.zw, r3.xyxy, r3.yxyx
```

```
  dp3 r3.w, r3.xyy, r3.xyy
  + adds r3.z, r3.zw
  mad r3.xyz, r3, c2.x, c2.y

  // calculate new half-level coordinates
  mad r1.xy, r3, r1.z, r1.xy

  serialize

  // scale the index to the correct range, and round
  mad r0.z, r4.w, c2.z, c2.w

  // per-level rescaling
  mul r4.xyz, r4, c8.z
  + floors r0.z, r0.z

  // add contribution
  dp3 r4.w, r4, r3
  + sgts r0.w, r0.z
  add r2.z, r2.z, r4.w

  // half size of quarter-level difference
  + mulsc r1.z, c1.w, r1.z

  add r2.x, r2.x, r0.z

  // Abort?  Will abort if fetch points to (0, 0)
  + setp eq r0. , r0.z

  // abortion jump
   (p0) jmp L1

  // point to selected node in texture
  mad r2.x, r2.x, r0.w, r3.w
////////////////////////////////////////////////
// to 1x1
////////////////////////////////////////////////

  // fetch coefficient
  tfetch1D r4.xyz , r2.x, tf0, MagFilter=point, MinFilter=point,
      UnnormalizedTextureCoords=true, FetchValidOnly=false // from 8:8:8:8
  // don't need to fetch index

  // find out which quadrant we're in, and build the basis
  sge r3.xy, r0.xy, r1.xy
  sgt r3.zw, r3.xyxy, r3.yxyx
  dp3 r3.w, r3.xyy, r3.xyy
  + adds r3.z, r3.zw
  mad r3.xyz, r3, c2.x, c2.y

  serialize

  // per-level rescaling
  mul r4.xyz, r4, c8.w

  // add contribution
  dp3 r4.w, r4, r3
  add r2.z, r2.z, r4.w


////////////////////////////////////////////////
       label L1
////////////////////////////////////////////////
  alloc colors
  exece

  mov oC0.xyz, r2.z
  + movs oC0.w, c0.x
```

***Listing A1.*** *Xbox 360™ microcode shader for texture decompression.*

```
  xps 3 0
  config AutoSerialize=false

  def c0, 16777215.0, 0.5, 1.0, 2.0
  def c1, 0.0002, 256.0, 65535.0, 0.0
  def c2, 1.386294361, 0.0, 0.0, 0.0
  def c3, 127.0, 127.5, 0.0, 0.0
  def c4, 0.8192, 0.00390625, -254.0, 4.0

  dcl texcoord0 r0.xy
  dcl_texcoord1 r1.xyz


  // this is so that if "FetchValidOnly" says don't fetch, we're pointing at the root
  // node
  mov r1.xy, r0
  + movs r0.x, c1.w

  // fetch transport function offset for this texel
  tfetch2D r0.x1  , r1.xy, tf0, MinFilter=point, MagFilter=point,
      FetchValidOnly=true
  serialize

  // is it an invalid pixel?
  sgts r5.x, r0.x

  // rescale to unnormalized integer
  mad r0.x, r0.x, c0.x, c0.y
  floor r0.x, r0.x

  // fetch base scale from transport & environment texture
  tfetch1D r1.yx  , r0.x, tf1, MinFilter=point, MagFilter=point,
     UnnormalizedTextureCoords=true, FetchValidOnly=true
  tfetch1D r1.  yx, r0.y, tf2, MinFilter=point, MagFilter=point,
     UnnormalizedTextureCoords=true, FetchValidOnly=true
tfetch1D r0.___w, r0.y, tf2, MinFilter=point, MagFilter=point,
     UnnormalizedTextureCoords=true, FetchValidOnly=true
  serialize


  // rescale and multiply base scales
  mad r1, r1, c0.y, c0.y
  mad r1.xy, r1.yw, c4.y, r1.xz
  mul r1.w, r1.x, r1.y


  // move on to root nodes
  add r0.xy, r0, c0.z


////////////////////////////////////////////////////////////////////////////
  label L0

  // fetch current nodes
  tfetch1D r2.xyzw, r0.x, tf1, MinFilter=point, MagFilter=point,
     UnnormalizedTextureCoords=true, FetchValidOnly=true
  tfetch1D r3.xyzw, r0.y, tf2, MinFilter=point, MagFilter=point,
     UnnormalizedTextureCoords=true, FetchValidOnly=true

  // calculate current level scalar
  mad r0.z, r0.w, c4.z, c0.w
  mul r0.z, r0.z, c0.y
  exp r0.z, r0.z

  serialize


  add r0.y r0, c0.z

  // scale by level scalar
```

```
mul r2.xyz, r2, r0.z
mul r3.xyz, r3, r0.z

add sat r4.x, r2.w, -r0.w
add sat r4.y, r3.w, -r0.w
+ movs r0.w, r2.w


// multiply and add to accumulating integral value
dp3 r1.z, r2, r3
+ muls r4.z, r4.xy
add r1.w, r1.w, r1.z
+ setp_gt r4._, r4.z
subs r4.w, r4.xy

// Now we need to check if we agree on the next node: if the first tree says the next
//  node is a child, but the second says a sibling or
//  a parent, we need to jump over the child of the first tree (and vice versa).

// case 1: next node in tree 1 is child, next node in tree 2 is child
//  here we need to carry on to the child, but omit both "jump" nodes
 (p0) add r0.xy, r0, c0.z


setp_gt r4._, r4.w


// case 2: next node in tree 1 is a child, next node in tree 2 is not
//  here we need to use the "jump" node to omit the branch starting at the first child
//  for tree 1
 (p0) exec
 (p0) tfetch1D r2.xyz , r0.x, tf1, MinFilter=point, MagFilter=point,
   UnnormalizedTextureCoords=true, FetchValidOnly=true, OffsetX=1.0
serialize
 (p0) mad r2.xy, r2, c3.x, c3.y
 (p0) floor r2.xy, r2
 (p0) mad r2.x, r2.y, c1.y, r2.x
 (p0) add r0.x, r0.x, r2.x
+ (p0) movs r0.w, r3.w

exec
setp_gt r4._, -r4.w

// case 3: next node in tree 1 is not a child, next node in tree 2 is
//  here we need to use the "jump" node to omit the branch starting at the first child
//  for tree 2
 (p0) exec
 (p0) tfetch1D r3.xyz_, r0.y, tf2, MinFilter=point, MagFilter=point,
   UnnormalizedTextureCoords=true, FetchValidOnly=true, OffsetX=1.0
serialize
 (p0) mad r3.xy, r3, c3.x, c3.y
 (p0) floor r3.xy, r3
 (p0) mad r3.x, r3.y, c1.y, r3.x
 (p0) add r0.y, r0.y, r3.x

exec


// If we're at the end node, try to jump out of the loop.  Of course, we can only do
//  this if all 64 pixels in the vector have reached
//  their end node.  So if we can't jump out we need to perform an 'idle' loop until
//  the whole vector is finished.
mul r2.w, r2.w, r3.w
mul r2.w, r2.w, r5.x

setp_eq r2._, r2.w
 (p0) jmp L1

// move onto the next node, provided we're not at the root
 (!p0) add r0.xy, r0, c0.z
```

```
  (p0) mul r0.xy, r0, c1.w

jmp L0
////////////////////////////////////////////////////////////////////////////

label L1

alloc colors
exece

mul oC0.xyz, r1.w, c255.x
+ movs oC0.w, c0.z
```

***Listing A2.*** *Xbox 360™ microcode shader for double product integration.*

## 4.8 Acknowledgements

I'd like to thank Ash Henstock for helping me make figure 7. Additionally Kieran Connell and Tom Grove for kindly permitting the use of their photograph in figure 5.

## 4.9 References

[Hu08] HU, YAOHUA. 2008. *Lightmap compression in Halo 3*. Presentation, Game Developer Conference, San Francisco, CA, February 2008

[GHB*05] GORSKI, K. M., HIVON, E., BANDAY, A. J., WANDELT, B. D., HANSEN, F. K., REINECKE, M., AND BARTELMANN, M. 2005.. *HEALPix: A framework for high-resolution discretization and fast analysis of data distributed on the sphere*. The Astrophysical Journal, 622:759-771, April 2005.

[SDS95] STOLLNITZ, E. J., DEROSE, T.D. AND SALESIN, D.H. 1995. *Wavelets for computer graphics: A primer (part 1)*, IEEE Computer Graphics and Applications, Vol. 15, pp. 76-84.

[DAUBECHIES92] DAUBECHIES, I. 1992. *Ten lectures on wavelets*. SIAM publishing, 978-0898712742.

[NRH03] NG, R., RAMAMOORTHI, R. AND HANRAHAN, P. 2003. *All-frequency shadows using non-linear wavelet lighting approximation.* ACM transactions on graphics (Siggraph 2003 Proceedings), pp. 376-381, San Diego, CA.

[NRH04] NG, R., RAMAMOORTHI, R. AND HANRAHAN, P. 2004. *Triple-product wavelet integrals for all-frequency relighting.* ACM transactions on graphics (Siggraph 2004 Proceedings), pp. 477-487, Los Angeles, CA, August 2004.

[HLS07] HORMANN, K. , LÉVY, B. AND SHEFFER, A. 2007. *Mesh parameterization: Theory and practice.* ACM SIGGRAPH Course 27 course notes, San Diego, CA, August 2007.

[DCH05] DIVERDI, S., CANDUSSI, N. AND HÖLLERER, T. 2005. *Real-time rendering with wavelet-compressed multi-dimensional datasets on the GPU,* Technical Report 2005-05, UCSB.

[MHL*06] MA, W.-C., HSIAO, C.-T., LEE, K.-Y., CHUANG, Y.-Y. AND CHEN, B.-Y. 2006. *Real-time triple product relighting using spherical local-frame parameterization.* The Visual Computer, Vol. 22, No 9-11, pp. 682-692. .

[WANWONG07] WAN, L. AND WONG, T.-T. 2007. *Sphere maps with the near-equal solid angle property.* Presentation, Game Developer Conference (GDC2007), San Francisco, CA, March 2007. http://www.cse.cuhk.edu.hk/~ttwong/papers/spheremap/spheremap.html

[SCHRÖDERSWELDENS95] SCHRÖDER, P. AND SWELDENS, W. 1995. *Spherical wavelets: Efficiently representing functions on the sphere.* In Proceedings of SIGGRAPH 1995, ACM Transactions on Graphics, pp. 161-172, Los Angeles, CA, August 1995.

[SLOAN08] SLOAN, P.-P.. 2008. *Stupid spherical harmonics tricks.* Presentation, Game Developer Conference (GDC2008), San Francisco, CA, February 2008. http://www.ppsloan.org/publications/StupidSH35.pdf

[FUJITAKANAI04] FUJITA, M. AND KANAI, T. 2004. *Precomputed radiance transfer with spatially-varying lighting effects.* In Proceedings of Proceedings of the International Conference on Computer Graphics, Imaging and Visualization (CGIV 2004), pp. 101-108..

[RAMAMOORTHIHANRAHAN01] RAMAMOORTHI, R. AND HANRAHAN, P. 2001. *An efficient representation for irrandiance environment maps.* ACM transactions on graphics (Siggraph 2001 Proceedings), pp. 497 – 500, Los Angeles, CA, August 2001.

# Chapter 5


# Effects & Techniques

Dominic Filion[1]
Rob McNaughton[2]

BLIZZARD ENTERTAINMENT

[1] dfilion@blizzard.com
[2] rmcnaughton@blizzard.com

**Figure 1**. *A screenshot from* StarCraft II

## 5.0 Abstract

In this chapter we present the techniques and algorithms used for compelling storytelling in the context of the *StarCraft II*© real-time strategy game. We will go over some of the design goals for the technology used to empower our artists for both in-game and "*story mode*" settings as well as describe how the Blizzard art style influenced the design of the engine. Various aspects of our lighting pipeline will be unveiled, with a strong focus on several techniques making use of deferred buffers for depth, normals, and coloring components. We will show how these deferred buffers were used to implement a variety of effects such as deferred lighting, screen-space ambient occlusion and depth of field effects. Approaches with respect to shadows will also be discussed.

## 5.1 Introduction

*Starcraft II* presented unique challenges for development as a second installment in a well-known franchise with the first entry dating from ten years before. During the process of development we had to overcome these and we will share the engineering

choices that were made in the context of the *Starcraft II* graphics engine, and in particular how these choices were made to support the unique Blizzard art style.

## 5.2 Engine Design Goals

Early on during development, we had several themes in mind that we wanted to drive the development of our graphics engine with:

*Scalability First*

A major design goal is the scalability of the engine. Blizzard games are well known for their ability to support a range of consumer hardware, including fairly outdated specifications, while always providing a great player experience. This was typically achieved in our previous games by keeping the playing field pretty even between all players, with a minimal set of video options. For *Starcraft II*, we wanted to maximize compatibility with less capable systems to ensure hassle-free game play for as broad a player base as possible. Yet we also wanted to utilize the full potential of any available hardware to ensure the game's looks were competitive. This meant supporting a wide range of hardware, from ATI Radeon™ 9800/NVIDIA GeForce™ FX's to the ATI Radeon™ HD 4800s and NVIDIA GeForce™ G200s, targeting maximum utilization on each different GPU platform.

This scalability translated in a fairly elaborate shader framework system. A full discussion of our shader system could very well encompass a whole chapter by itself so we will not focus on it here, so this is only a short overview.

We only had one or two artists that would actually have any interest in writing their own shaders so early on, instead of focusing on writing elaborate shader prototyping tools for artists we decided to focus our efforts on making the shader framework as flexible and easy to use for programmers as possible, which is somewhat counter to the industry trend at this time.

Thus, writing shader code in our game is generally very close to adding a regular C++ file in our project. Figuratively, we treat the shader code as an external library called from C++, with the shader code being a free form body of code organized structurally as a C++ codebase would. Thus, the concept of shaders can be a loose one in *Starcraft II* – the shader code library defines several entry points that translate to different shaders, but one entry point is free to call into any section of shader code. Thus it would be difficult to talk about how many "individual" shaders *Starcraft II* uses, as it is a single body of code from which more than a thousand shader permutations will generally be derived for a single video options configuration. Making our shader framework system as familiar as possible for programmers has enabled us faster turnaround time when debugging shaders. In our case, when our technical artists have some ideas for new

shaders they will generally prototype it using 3D Studio MAX renders and a programmer will implement an optimized version of the effect, often times adding reusable abstractions to the shader that may not have been apparent to an artist. In all, *Starcraft II* uses about 8,000 unique, non-repeating lines of shader code, divided amongst 70 files. Interestingly enough, we've come to the point where the body of shader code alone has grown larger than the entire codebase for games from the early stages of the games industry.

*Stress GPU over CPU*

We chose early on to stress the GPU more than the CPU when ramping up quality levels within the game. One of the main reasons for this is that, in *Starcraft II*, you are able to spawn and manage potentially hundreds of the smaller base units such as *zerglings* and *marines*. In large-scale multiplayer games with eight players, this can translate to up to roughly five hundred units on the screen at a single time, at peak. Because the number of units built is largely under the control of the player (as well as due to the choice of the selected race), balancing the engine load such that CPU potential is well utilized in both high-unit count and low-unit unit count situations becomes cumbersome.

**Figure 2**. *Players can control a very high number of units, thus balancing batch counts and vertex throughputs is key to reliable performance.*

In contrast, GPU loads in our case tend to be much more affected by the pixel shader load, which tends to stay constant for a real-time strategy game because of the generally low overdraw. Although vertex counts for a particular scene can ramp up well above a million in a busy battle scene, this type of vertex throughput is well handled by most modern GPU hardware. This has driven us to push the engine and art styles towards ever increasing pixel-shader level effects whenever possible, minimizing batch counts and using relatively conservative vertex counts.

The *Starcraft II* units are generally viewed from a good distance away and therefore result in a small screen-space footprint during game play. Therefore, increasing the model vertex counts would have given us marginal benefits during actual game play in most cases. For that reason we avoided generating art assets with large polygon counts.

*Dual Nature of the Engine*

*Starcraft II* is supported by an engine that in many ways has a split personality; during normal game play we typically render scenes from a relatively far away distance, with high batch counts, and a focus on action rather than details. At the same time, we really wanted to push our storytelling forward with *Starcraft II*, and this is where the game's "*Story Mode*" comes in. In this mode, the player generally sits back to take in the game's rich story, lore and visuals, interacting with other characters through dialogues and watching actions unfold. This is the mode where most of *Starcraft II*'s storytelling happens and it has radically different and often opposing constraints to the in-game mode; story mode generally boasts lower batch counts, close-up shots, and a somewhat more contemplative feel – all things more typical of a first person shooter.

**Figure 3**. *In-game view vs. "story mode" in Starcraft II.*

We will explore how these different design goals affected our algorithmic choices. We will showcase some of our in-game cinematics and peel off some of the layers of technology behind it, as well as shows examples of the technology in actual gameplay.

## 5.3 Screen-Based Effects

One of the objectives for *Starcraft II* was to present rich lighting environments in story mode, as well as more lighting interactions within the game itself. Previously in *Warcraft III*, each unit had a hard limit as to the number of lights that could affect it at any given time. This would cause light popping as units moved from one of light influences to another. For that reason, the use of dynamic lighting was fairly minimal. At the same time, using forward rendering approaches, we were quickly presented with the problem of the large increase in the number of draw call batches. One of the stress cases we used was a group of marines, with each marine casting a flickering light that in turn shades the surrounding marines around him. The batch counts in such cases rapidly became unmanageable, and issues were also present with our deeply multi-layered terrain rendering, thus requiring complex terrain slicing and compounding the problem.

The goal to reduce batch counts was the first step that sent us on the road towards using deferred buffers for further treatment at the end of the frame. "Deferred rendering" term (as covered in [CALVER04], [SHISHKOVTSOV05] and in [KOONE07]) often refers to the storage of normals, depth and colors to apply lighting at a later, "deferred" stage. In our case we use it in the broader sense to mean any graphical technique which uses this type of deferral, even in cases where a non-deferred approach would have been viable as well. We'll explore issues and commonalities with all things deferred.

In practice, we've found many pros and cons for using deferred computations with the storage of relevant information, such as depth values, normals, etc. Although storage of this information consumes memory and bandwidth, as well as the extra cost of re-sampling the buffers at the processing stage, it generally helps greatly with scalability by ensuring the cost curve of effects tend to stay linear instead of exponential. When using deferred buffers, adding more layers of effects generally results in a linear, fixed cost per frame for additional full-screen post-processing passes regardless of the number of models on screen, whereas the same effect with forward rendering exhibits an exponential cost behavior. This leads us unto our goal of maximizing resource usage regardless of whether there five or five hundred units on the screen.

Keeping batch counts low is paramount for any RTS game. We were thus naturally drawn away from trying to fill up the deferred buffers in a separate pass and instead relied on multiple render targets (MRTs) to fill the deferred buffers during the main rendering. Thus, in the main rendering pass, several MRTs will be bound and filled with their respective information. Much of the mainstream hardware is limited to four

render targets bound at once, so right now it makes sense to pick deferred component that will fit within the sixteen individual channels this provides for.

For *Starcraft II*, any opaque model rendered will store the following to multiple render targets bound in its main render pass:

- Color components not affected by local lighting, such as emissive, environment maps and forward-lit color components;
- Depth;
- Per-pixel normal;
- Ambient occlusion term, if using static ambient occlusion. Baked ambient occlusion textures are ignored if screen-space ambient occlusion is enabled;
- Unlit diffuse material color;
- Unlit specular material color.

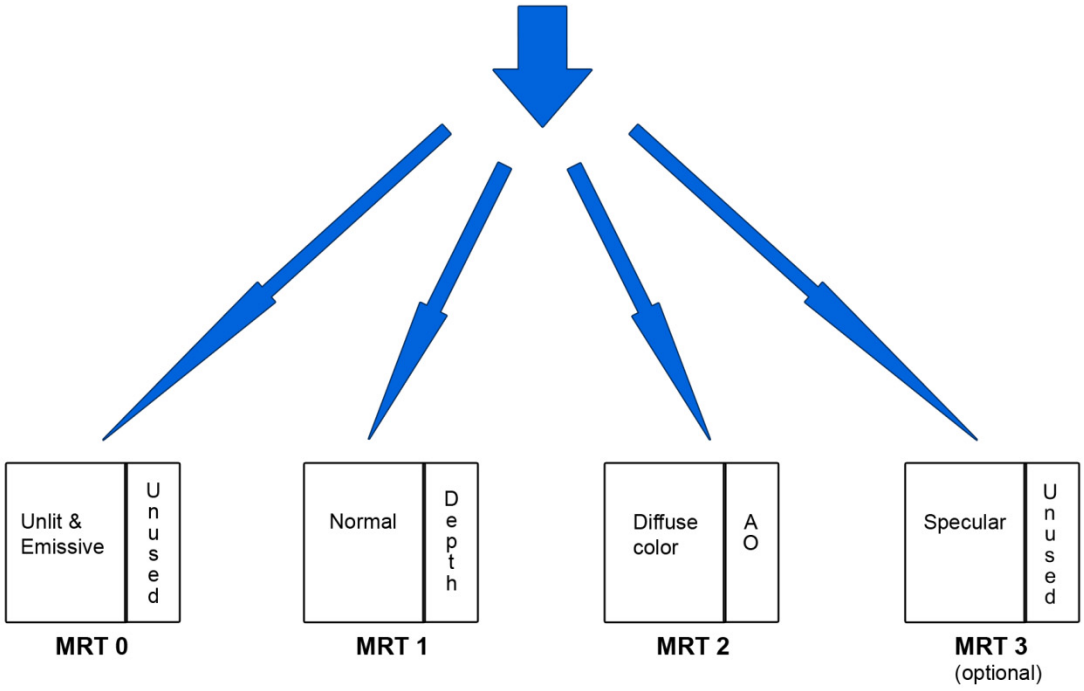


**Figure 4**. *MRT setup.*

As a quick reminder, simultaneously bound render targets under DirectX® 9 need to be of the same width and height, and in most cases, the same bit depth. Some, but not all, hardware supports independent color write control on each MRT which should be used whenever possible to minimize wasted bandwidth for unused components. For example, the alpha component is not used on all of these buffers.

We are making heavy use of HDR ([RWPD05])in *Starcraft II*, and thus all these buffers will normally be four-channel 16-bit floating point formats. Using higher precision format helps to sidestep accuracy issues and minimizes pixel shader instructions for

decoding the buffers. The restriction that all buffers should be of the same bit depth unfortunately forces the use of much wider buffers than we actually need for a many of these buffers, pushing the output bandwidth to 24 bytes per pixel. We've found however that even with this very heavy bandwidth cost the freedom this allows us in the placement of our lighting was well worth the expense.

In the majority of cases, all objects outputting to these buffers are opaque, with some notable exceptions; we will come back to the problem of handling rendering transparent objects later.

Our terrain is multi-layered, in which case the normals, diffuse and specular colors are blended in these buffers, while leaving the depth channel intact – only the bottom terrain layer writes to the depth buffer.

Rendering to the MRTs provides us with per-pixel values that can be used for a variety of effects. In general, we use:

- *Depth values* for lighting, fog volumes, dynamic ambient occlusion, and smart displacement, depth of field, projections, edge detection and thickness measurement.
- *Normals* for dynamic ambient occlusion
- *Diffuse and specular* for lighting

## 5.4 Deferred Lighting

Deferred lighting in *Starcraft II* is used for local lights only: point and spot lights with a defined extent in space. Global directional lights are forward rendered normally; because these lights have no extents and cover all models, there is little benefit in using deferred rendering methods on them, and it would actually be slower to resample the deferred buffers again for the entire screen.

The computation results for deferred rendering vs. forward rendering equations are equivalent, yet the deferred form is more efficient for complex lighting due to the tighter light coverage offered by rendering the light shapes as a post-process. For the most part, the new equation simply moves terms from one stage of the rendering pipeline to a later stage, with the notable exception of the pixel's view space position, which is more efficiently reconstructed from the depth information.

**Pixel Position Reconstruction**

Pixel shader 3.0 offers the VPOS semantic which provides the pixel shader with the x and y coordinates of the pixel being rendered on the screen. These coordinates can be normalized to the [-1..1] range based on screen width and height to provide a normalized eye-to-pixel vector. Multiplying by the depth will provide us with the pixel's

view space position. For pixel shader 2.0, a slightly slower version is used where the vertex shader feeds to the pixel shader a homogeneous equivalent to VPOS, from which the w component must be divided in the pixel shader.

```
float3 vViewPos.xy = INTERPOLANT VPOS * half2( 2.0f, -2.0f ) + half2( -1.0f, 1.0f ) ) *
                     0.5 * p vCameraNearSize * p vRecipRenderTargetSize;

vViewPos.zw  = half2( 1.0f, 1.0f );
vViewPos.xyz = vViewPos.xyz * fSampledDepth;

float3 vWorldPos = mul( p_mInvViewTransform, vViewPos ).xyz;
```

**Listing 1**. *Pixel position reconstruction*

### Stenciling, Early-Z and Early-Stencil

In our case, usage of early stencil out provided substantial speed improvements for our scenes. Stenciling allows us to discard pixels that are covered by the light shapes but whose depth is too far behind the light to be affected by it.

By the same token, pixels that are in front of the light's influence will automatically benefit from early-z out as the light shape pixels will be discarded early on.

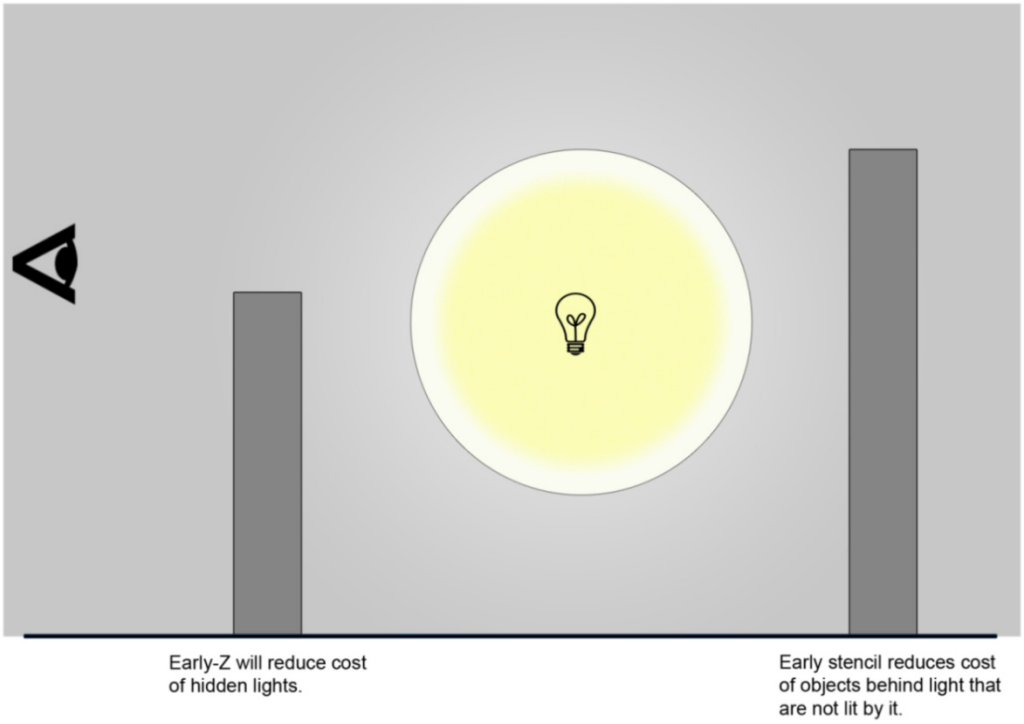


**Figure 5**. *Deferred rendering interaction with z-buffer and stencil buffer.*

To benefit from early stencil, the light shapes must be rendered again with color writes off before its normal lighting pass. The stencil operation is slightly different depending on whether the viewing camera sits inside the light shape – this has to be tested on the CPU.

If the camera is outside the light shape, effectively we want to only pass through pixels where only one side of the light shape is visible (which will always be the front facing parts); this implies the pixel of the surface being lit is hiding the back facing part of the light shape, aka the surface pixel is inside the light shape. This can be achieved by clearing the stencil and passing any pixels that show a front face but no back face for the light shape.

If the camera is inside the light shape, then only the back side of the light shape is visible. In this case we want to color any of the light shape's back facing pixels that failed the z-test, which implies there is a lit surface inside the light shape.

**Deferred Lighting in Action**

The low batch counts of the deferred system allowed us to truly refine the lighting in our cinematic scenes. Here we see a shot of one of our scenes, which consists of roughly x (fifty) dynamic lights in the same room, from small Christmas lights to larger lighting coming from the TV screen.


**Figure 6**. *Deferred lighting allows us to use complex lighting environments.*

## 5.5 Screen-Space Ambient Occlusion

### SSAO Basics

Although our initial interest in screen space ambient occlusion was sparked by the excellent results that we observed in such games as Crysis©, we arrived at our solution independently. In a nutshell, the main idea behind screen space ambient occlusion is to approximate the occlusion function at points on visible surfaces by sampling the depth of neighboring pixels in screen space. The resulting solution will be missing occlusion cues from objects that are currently hidden on the screen, but since ambient occlusion tends to be a low frequency phenomenon, the approximation is generally quite convincing.


**Figure 7**. *Scene with lighting information only; softer shading cues are accomplished through SSAO.*

SSAO requires depth to be stored in a prior step; for *Starcraft II* this is the same depth buffer that we use for deferred lighting. The ambient occlusion term that will be produced by the SSAO is stored in the alpha channel of our diffuse deferred render buffer, which is then used to modulate lighting from certain global and local lights.

At any visible point on a surface on the screen, multiple samples (8 to 32) are taken from neighboring points in the scene. These samples are offset in 3D space from the current point being computed; they are then projected back to screen space to sample the depth at the sample location.

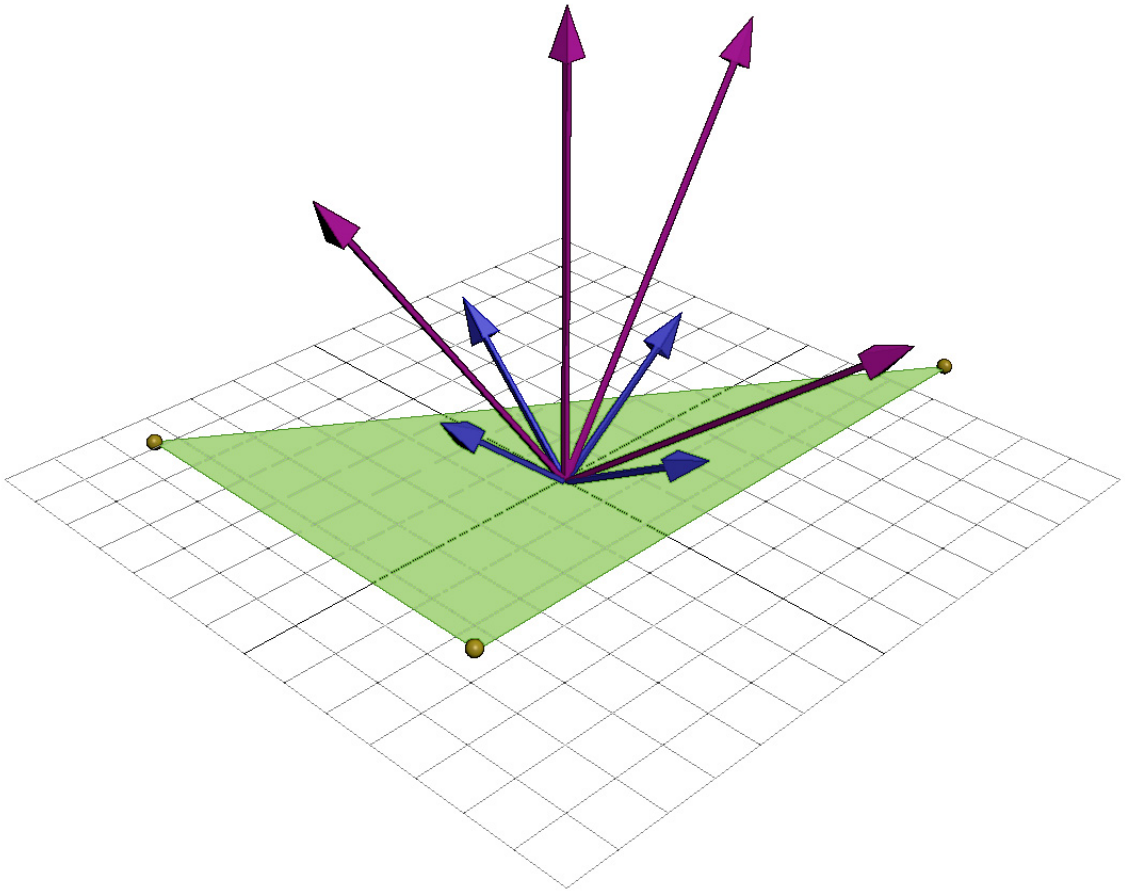

**Figure 8.** *Overview of SSAO sampling process.*

The objective is to check if the depth sampled at the point is closer or further away than the depth of the sample point itself. If the depth sampled is closer, than there is a surface that is covering the sample point. In general this means there is another surface nearby that is covering the area in the vicinity around the current pixel, and some amount of occlusion should be occurring. It is the average of these samples that will determine the total occlusion value for the pixel.

The depth test of the sample point with its sampled depth is not simply a Boolean operation. If the occluding surface is very close to the pixel being occluded, it will occlude a lot more than it is further away; and in fact beyond a certain threshold there needs to be no occlusion at all as we don't want surface far away from the surface to occlude it. Thus we need some type *occlusion function* to map the relationship between

the depth delta between the sample point and its sampled depth, and how much occlusion occurs.

If the aim is to be physically correct, then the occlusion function should be quadratic. In our case we were more concerned about being able to let our artists adjust the occlusion function, and thus the occlusion function can be arbitrary. The occlusion functions can be any function that adheres to these criteria:

- Negative depth deltas should give zero occlusion (the occluding surface is behind the sample point);
- Smaller depth deltas should give higher occlusion values;
- The occlusion value needs to fall to zero again beyond a certain depth delta value.

For our implementation we simply chose a linearly stepped function that is entirely controlled by the artist. There is a *full occlusion* threshold where every positive depth delta smaller than this value gets complete occlusion of one, and a *no occlusion* threshold beyond which no occlusion occurs. Depth deltas between two extremes fall off linearly from one to zero, and the value is exponentially raised to a specified *occlusion power* value. If a more complex occlusion function is required, it can be pre-computed in a small 1D texture to be looked up on demand.

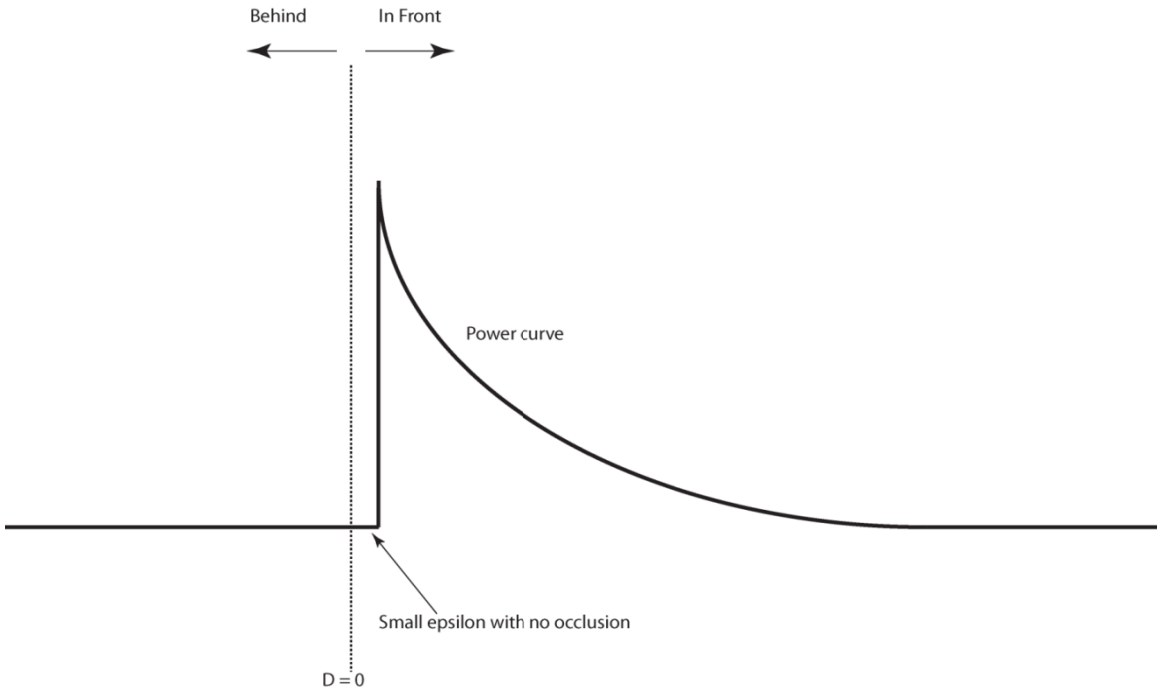


**Figure 9**. *Occlusion function.*

To recap, the SSAO generation process is:

- Compute the view space position of a pixel;

- Add (8 to 32) offset vectors to this position;
- Remap these offset vectors to where they are in screen space;
- Compare the depth of each offset vector with the depth at the point where the offset is;
- Each offset contributes occlusion if the sampled depth on screen is behind the depth of the offset vector. Occlusion factor is based on the difference between the sampled depth and the depth of the offset vector.

**All in the Details: SSAO Artifacts**

The basics of SSAO are fairly straightforward but there several alterations that need to be made for it to produce acceptable results.

*Sampling Randomization*

To smooth out the results of the SSAO lookups, the offset vectors must be randomized thoroughly. One good approach, as given in (6) is to generate a 2D texture of random normal vectors and lookup this texture in screen space, thus retrieving a unique random vector per pixel on the screen. More than one random vector must be generated per pixel however: these are generated by passing a set of offset vectors in the pixel shader constant registers and reflecting these vectors through the sampled random vector, thus generating a semi-random set of vectors at each pixel. The set of vectors passed in as registers is not normalized – having varying lengths helps to smooth out the noise pattern. In our case we randomize the length of the offset vectors from 0.5 to 1, avoiding samples clustered too close to the source point. The lengths of these vectors are scaled by an artist-controlled parameter that determines the size of the sampling area.

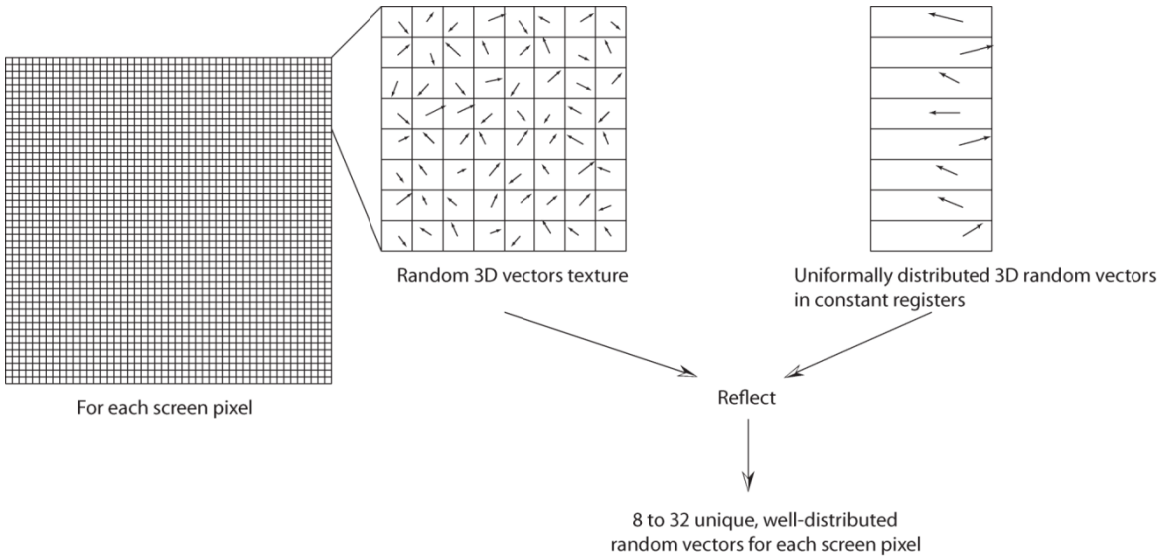


**Figure 10**. *Randomized sampling process.*

**Figure 11.** *SSAO term after random sampling applied. Applying blur passes will further reduce the noise to achieve the final look.*

*Blurs*

The previous step helps to break up the noise pattern, producing finer grained pattern that is less objectionable. With wider sampling areas however, more processing such as blurs become necessary. The ambient occlusion results are low-frequency, and we've found losing some of the high-frequency detail due to blurring was not an issue.

The ambient occlusion must not bleed through edges to objects that are physically separate within the scene, and thus a form of smart Gaussian blur is used. This blur samples the nearby pixels as a regular Gaussian blur shader would, yet the normal and depth for each of the Gaussian samples is sampled as well (encoding the normal and depth in the same render targets presents significant advantages here). If either the depth from Gaussian sample differs from the center tap by more than a certain threshold, or the dot product of the Gaussian sample and the center tap normal is less than a certain threshold value, then the Gaussian weight is reduced to zero. The sum of the Gaussian samples is then renormalized to account for the missing samples.

Several blur passes can thus be applied to the ambient occlusion output to completely eliminate the grain pattern.

*Self-occlusion*

The output from this produces convincing results, but has the issue that in general no area is ever fully unconcluded due to the fact that the random offset vectors from each point will penetrate through the surface of the object itself, and the object becomes self-occluding at all times.

One possible solution around this is to generate the offset vectors around a hemisphere centered around the normal at that point on the screen (which implies a deferred normal buffer has to be generated, another strong point for reusing output from deferred rendering). Transforming each offset vector by a matrix can be expensive however, and one compromise we've used is to instead perform a dot product between the offset vector and the normal vector at that point, and to flip the offset vector is the dot product is negative. This is a cheaper way to solve the problem at the expensive of possibly making the noise pattern more predictable.

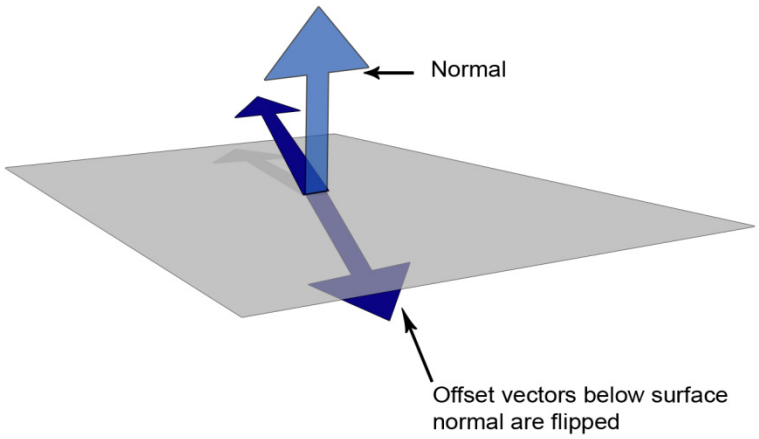


**Figure 12.** *Handling self-occlusion.*

*Edge Cases*

The offset vectors are in view space, not screen space, and thus in close-up camera shots these vectors will be longer so that the SSAO sampling area remains constant within the scene. This can mean more noise for close-ups, but also presents problems when samples end up looking up outside the screen. There is no straightforward way to deal with the edge case, as the depth information outside the screen is not present:

rendering a wider area than what is seen in screen could help improve this, but is not a robust solution that would work in all cases. The less objectionable way to deal with this is to ensure that samples outside the screen return a large depth value, ensuring they would never occlude any neighboring pixels. This can achieved through the "border color" texture wrapping state.

To prevent unacceptable breakdown of the SSAO quality in extreme close-ups, we've found it necessary to impose an SSAO screen-space sampling area limiter. In effect, if the camera is very close to an object and the SSAO samples end up being too wide, the SSAO area consistency constraint is violated so that the noise pattern doesn't become too noticeable. Another alternative would to simply vary the number of samples based on the sampling area size, but this can introduce wide frame rate swings that we have been eager to avoid.

**SSAO Performance**

SSAO can give a significant payback in terms of mood and visual quality of the image, but it can be quite an expensive effect. The main bottleneck of the algorithm is the sampling itself: the random nature of the sampling, which is necessary to minimize noise, wreaks havoc with the GPU's texture cache system and can become a problem if not managed. The performance of the texture cache will also be very dependent on the sampling area size, with wider areas straining the cache more and yielding poorer performance. Our artists quickly got in the habit of using SSAO to achieve a faked global illumination look that suited their purposes: this required more samples and wider sampling areas, so extensive optimization become necessary for us.

One method to bring SSAO to an acceptable performance level relies on the fact that ambient-occlusion is a low-frequency phenomenon. Thus we've found there is generally no need for the depth buffer sampled by the SSAO algorithm to be at full screen resolution. The initial depth buffer can be generated at screen resolution, since we reuse the depth information for other effects within the game and it has to fit the size of the other MRTs, but it is thereafter down sampled to a smaller depth buffer that is a quarter size of the original on each side. The down sampling itself does have some cost but the payback in improved texture cache afterwards is very significant.

**SSAO and Global Illumination**

*Wide-area SSAO vs. Outline Enhancement*

One thing we've quickly discovered is that our cinematic artists loved the quasi-global illumination feel of the output. If the sampling area of the SSAO is wide enough, the look of the scene changes from darkness in nooks and crannies to a softer, ambient feel.

The SSAO implementation was thus pulled by the art direction into two somewhat conflicting directions: on the one hand, the need for tighter, high-contrast occluded zones in deeper recesses, and on the other hand, the desire for the larger, softer ambient look of the wide area sampling.

In our case, we resolved this conundrum by splitting the SSAO samples between two different sets of SSAO parameters: one quarter of our samples are concentrated in a small area with a rapidly increasing occlusion function, while the remainder uses a wide sampling area with a gentler function slope. The two sets are averaged independently and the final result uses the value from the set which produces the most (darkest) occlusion.

## 5.6 Depth of Field

**Figure 13**. *Depth of field as used in Starcraft II for a cinematic feel.*

Because deferred rendering had to use generate a normal and a depth buffer every frame, we were naturally drawn to leverage this as much as possible. One of the post-processing effects crucial for many of our story mode scene has been depth of field. There has been a fair amount of interest in real-time depth of field rendering, as covered in [ScheuermannTatarchuk04] and [Demers05], for example. For storytelling we will often focus the camera on a specific object to attract the attention of the viewer. Here is a review of the problems we faced and how they were solved.

*Circle of Confusion*

The ingredient needed to perform the depth of field, which is common for real-time depth of field algorithm, is to compute a blur factor, or circle of confusion (CoC) for each pixel on the screen. We only are loosely to the definition of the circle of confusion, which would be the radius of the circle over which pixels, or rays, are blurred – the only point of importance is that higher levels of CoC needs to map to a blurrier image in some kind of predictable gradient.

Thus, the circle of confusion is art-driven instead of physics-driven. Artists specify a reference point that is a certain distance in front of the viewing camera. From this point, a *full focused* distance is specified – all depths whose distance from the reference point, either in front or behind the reference point, is less than this distance are sharp. Beyond this distance, the image progressively gets blurrier until it reaches the *fully unfocused* distance where the blurriness is maximal.

Thus computing the circle of confusion comes down to sampling from the deferred depth buffer and going through the following equation:

$$saturate\left(\frac{DofAmount \times max(0, Depth - FocalDepth - NoBlurRange)}{MaxBlurRange - NoBlurRange}\right)$$

*Equation 1. CoC as a function of depth.*

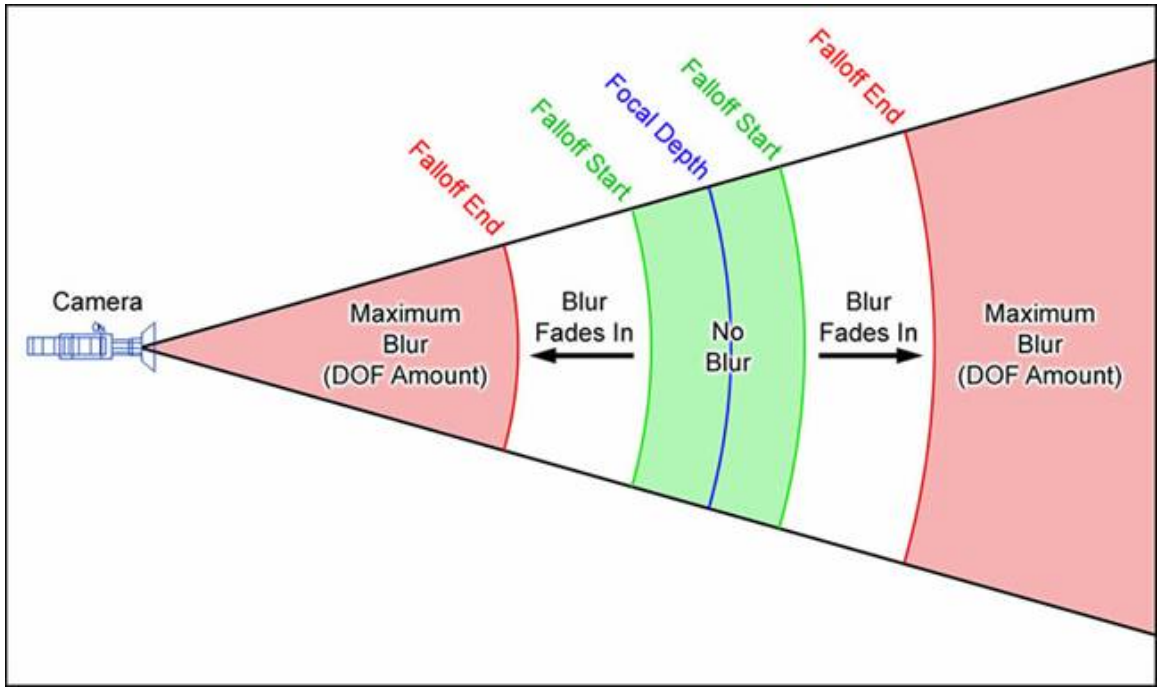


**Figure 14.** *Depth of field regions.*

*Blurring*

The circle of confusion value, for each pixel, can then be used to gradually blur the image. The blur needs to be gradual as the CoC value increases, and not present under-sampling artifacts.

Simply varying the width of the Gaussian filter kernel to simulate different blur levels doesn't work too well; larger blur factors require more blur samples to look as good. This is achievable by varying the number of samples per pixel, which unfortunately requires use of dynamic branching in the pixel shader, whose market penetration wasn't as high as we needed it to be.

Thus a different approach is to generate pre-blurred images at fixed levels and linearly interpolate the outputs from the two closest pre-blurred sample points. In practice, this means blurring between four different reference images: the sharp image, two other screen-sized images of increasing blurriness, and a maximal blur image with each side a quarter of the original.

These four images can be generated, and the depth of field image effects can then compute the CoC factor and use it to linearly blend between two of the images.

*Edge Cases*

The two previous steps work well to create the effect of blurriness in different area of the screen based on the pixel depth, but edges pose problems that require special care. To be convincing, depth of field needs to exhibit the following behaviors:

- Out-of-focus foreground objects should bleed unto objects behind them, whether sharp or unfocused;
- Conversely, bleeding of unfocused background objects unto foreground objects should be avoided – clearly visible, focused edges need to stay sharp.

However, the blurring process occurs by *looking up* samples from neighboring pixel and doing a weighted, which requires us to restate these statements in a way that maps better to how the algorithm is actually implemented, as follows:

- *Avoiding sharp halos:* Sharp pixels should not contribute to the weighted sum of any neighboring pixels or they will create halos as well - partially unfocused pixels should contribute more to the sum of neighboring pixels than more focused ones.
- *Bleeding of blurry objects over background:* the blurriness of a single pixel does not depend on the circle of confusion of that pixel alone; a nearby blurry pixel with a large circle of confusion would have the effect of making the current pixel blurry;
- *Depth ordering of blurriness:* Pixels in the background should not contribute to the weighted sum of pixels in the foreground;

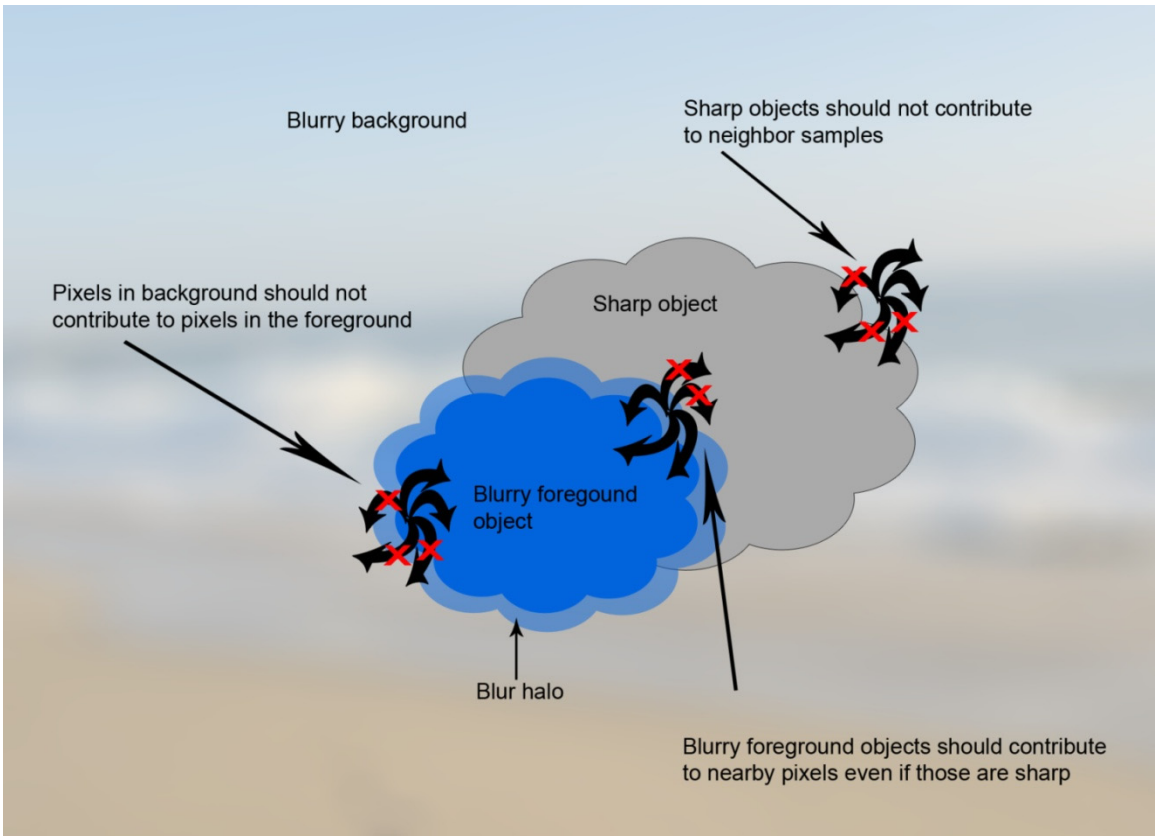


**Figure 15.** *Depth of field rules.*

For the first constraint, we must weigh the Gaussian samples by the corresponding CoC factors for each pixel. The sampled CoCs are renormalized so that the sum of all sampled CoCs adds up to one and each sampled CoC is multiplied by the corresponding sample. Thus, a sharp pixel will contribute less (or not at all) to neighboring pixels that are blurry. Although the effect may not be correct in a strict sense, the most important goal here is to eliminate any changes in color for blurry pixels when completely focused, neighboring pixels are changing.

The second constraint can be solved by blurring the circle of confusion factors in an image a quarter size per side, ensuring that any blurry pixels will cause neighboring pixels to become blurry as well.

Blurring all CoCs will violate the third constraint however, causing background objects to blur over foreground ones – we need to conserve some sense of depth ordering of the blur layers. We can achieve this through the following process:

- Downscale and blur the *depth map* in an image a quarter size per side;
- Sample both the blurred and non-blurred depth maps;
- If the blurred depth is smaller (closer) than the non-blurred depth, use the CoC from the *blurred CoC map;*

- If the blurred depth is larger (further away) than the non-blurred depth, compute and use the CoC for the current pixel only (no CoC blurring).

This effectively compares the Gaussian-weighted average of the cluster of pixels around the current pixel and compares it with depth at that pixel only. If this average is further away than the current pixel, then other pixels in that area tend to be behind the current pixel and do not overlap it. Otherwise they are in front, and their CoC is expected to affect the current pixel's CoC (blur over it).

*Putting It All Together*

Putting all the constraints in place, we can put everything together using the following process:

- Perform a full-screen pass to **compute the circle of confusion** for each pixel in the *source* image and store the result in the alpha channel of a downscaled *CoC* image buffer a quarter of the size on each side;
- **Generate the medium blur image** by applying a RGB Gaussian blur with each sample weighted by the CoC on the *source* image;
- **Generate the max blur image** by downscaling the RGB of the source image into an image buffer a quarter of the size on each side – the CoC and large blur buffers can be the same since they use different channels;
- **Blur the max blur image** with the RGB samples weighted by the downscaled CoC. The alpha channel, which contains the CoC, also gets blurred here, but its samples are not weighted by itself.
- **Downscale and blur the depth map** into a *downscaled depth image* – it should be noted in our case we reuse the downscaled depth for our SSAO pass as well (but do not blur depth for the SSAO);
- Then apply the final depth of field shader, binding the source image, medium and large blur/blurred CoC image, the non-blurred depth map and the downscaled depth image to the shader. The depth of field shader:
    - Computes the *small* blur value directly in the shader using a small sample of four neighbor pixels;
    - Computes the CoC for that pixel (the downscaled CoC would not match);
    - Samples the non-blurred and blurred depth to compare them – use computed CoC if blurred depth is further away than non-blurred depth, otherwise use CoC value samples from blurred CoC image;
    - Calculate contribution from each of the possible blur images: source, computed small blur color, medium and large blur images based on the CoC factor from zero to one;
    - Sums the contribution of the small, medium and large blurs
    - Output the alpha to include the contribution of the source (no blur) image.

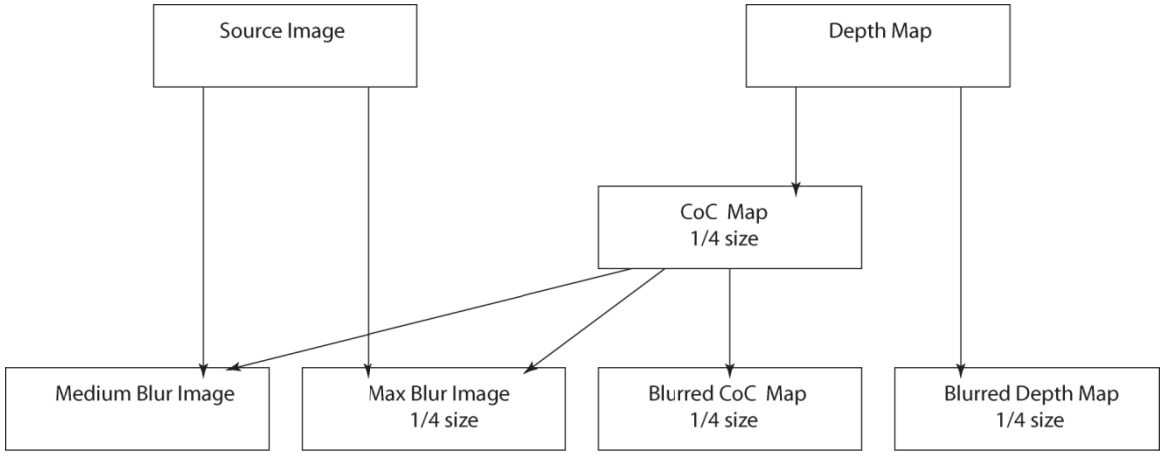


**Figure 15.** *Depth of field texture inputs & outputs. In the implementation some components reuse the same texture, but they're shown separated here for clarity.*

```
float       p_fDOFAmount;          // Depth of field amount.
float       p_fFocusDistance;      // Distance in focus.
float       p_fFullFocusRange;     // Range inside which everything is in focus.
float       p_fNoFocusRange;       // Range at which everything is fully blurred.
sampler2D   p_sMediumBlurMap;      // Medium blur map for depth of field.
sampler2D   p_sLargeBlurMap;       // Large blur map for depth of field.
sampler2D   p_sCoCMap;             // COC map.
sampler2D   p_sDownscaledDepth;

half ComputeCOC( float depth ) {
       return saturate( p_fDOFAmount * max( 0.0f, abs( depth - p_fFocusDistance ) -
       p_fFullFocusRange ) / ( p_fNoFocusRange - p_fFullFocusRange ) );
}

half4 Tex2DOffset( sampler2D s, half2 vUV, half2 vOffset ) {
    return tex2D( s, vUV + vOffset *
                  half2( 1.0f / p_vSrcSize.x, 1.0f / p_vSrcSize.y ) );}

half3 GetSmallBlurSample( sampler2D s, half2 vUV ) {
    half3 cSum;
    const half weight = 4.0 / 17;
    cSum = 0;
    cSum += weight * Tex2DOffset( s, vUV, half2( 0.5f, -1.5f ) ).rgb;
    cSum += weight * Tex2DOffset( s, vUV, half2( -1.5f, -0.5f ) ).rgb;
    cSum += weight * Tex2DOffset( s, vUV, half2( -0.5f, 1.5f ) ).rgb;
    cSum += weight * Tex2DOffset( s, vUV, half2( 1.5f, 0.5f ) ).rgb;
    return cSum;
}


// Depth of field mode.
half4 PostProcessDOF( VertexTransport vertOut ) {
    float4 vNormalDepth = tex2D( p_sNormalDepthMap, INTERPOLANT_UV.xy );
    float vDownscaledDepth = tex2D( p_sDownscaledDepth, INTERPOLANT_UV.xy ).a;
    half fUnblurredCOC  = ComputeCOC( PIXEL_DEPTH );

    half fCoC = tex2D( p_sLargeBlurMap, INTERPOLANT_UV.xy ).a;

    // If object is sharp but downscaled depth is behind, then stay sharp
    if ( vDownscaledDepth > vNormalDepth.a )
      fCoC = fUnblurredCOC;
```

```
    half d0 = 0.50f;
    half d1 = 0.25f;
    half d2 = 0.25f;
    half4 weights = saturate( fCoC * half4( -1 / d0, -1 / d1, -1 / d2, 1 / d2 ) +
                              half4( 1, ( 1 - d2 ) / d1, 1 / d2, ( d2 - 1 ) / d2 ) );
    weights.yz = min( weights.yz, 1 - weights.xy );

    half3 cSmall    = GetSmallBlurSample( p sSrcMap, INTERPOLANT UV.xy );
    half3 cMed      = tex2D( p sMediumBlurMap, INTERPOLANT UV.xy ).rgb;
    half3 cLarge    = tex2D( p sLargeBlurMap, INTERPOLANT UV.xy ).rgb;

    half3 cColor = weights.y * cSmall + weights.z * cMed + weights.w * cLarge;
    half fAlpha = dot( weights.yzw, half3( 16.0f / 17.0f, 1.0f, 1.0f ) );
    return half4( cColor, fAlpha );
};
```

**Listing 2.** *Depth of field shader.*

## 5.7 Dealing with Transparent Object Rendering

It's worthwhile to mention some of the issues with transparency, particularly with respect to the fact that deferred rendering typically do not support transparency very well, or at all. It should be noted that the challenges with transparent objects are not unique to deferred lighting, and in fact if one is striving for absolute correctness, transparency creates problems throughout the entire engine pipeline.

As is typical for any kind of deferred rendering techniques, transparencies are not taken into account by the deferred renderer. We've found however that lit transparencies don't contribute that significantly to the look of the scene as they will pick up the shade of the objects behind them. We do tag some transparent objects selectively as being affected by lighting, in which case these are forward rendered with a multi-pass method. We've been careful to avoid hard cap limits of any sorts on the engine whenever we can, and a multi-pass approach proved to be more scalable than a single-pass one.

Another advantage of the multi-pass lighting approach is that there is no need for more than a single shadow map buffer for the local lighting; the shadow map for each light can be applied one at a time and applied in succession. This proved important because the translucent shadow technology we've designed already needs a second shadow map to render the effect.

We've found that in our environments transparencies are not present in an amount sufficient to create very drastic lighting changes, except for a few specific pieces that we've tagged manually for forward rendering.

So, in light of this, we didn't strive to find clever solutions to fix problems with transparencies with respect to deferred lighting. Other subsystems however, can create

artifacts that are way more noticeable if absolutely no support for transparencies is present.

Depth of field, for instance, has very disturbing artifacts if depth is not taken into account for key transparencies. In this shot, we had a translucent egg that was showing as very crisp along its unfocused neighbors. SSAO also has issues with transparencies, especially for not-quite-opaque objects like bottles that effectively lose their ambient occlusion.

For some of these cases, using a simple layered system worked well enough for us. The depth map is first created from the opaque objects, opaque objects rendered, and depth-dependent post-processing effects subsequently applied on them. Transparent objects are then rendered as they would be normally from back to front, and key transparencies are allowed do a pre-pass where they will output their normals and depth to the deferred buffers. This effectively destroys the depth and normals information for the opaque objects behind these transparencies, but since all post-processing has been applied to these objects already, that information is no longer needed.

After the pre-pass, one more pass is used to update the ambient occlusion deferred buffer. The transparency itself is then rendered, and then another depth of field pass is done specifically on area covered by the transparency.

This certainly makes rendering that particular transparency much more expensive, and we only enable this option for key transparencies where maximum consistency is needed. In simpler cases, we will cheat by allowing the transparency to emit it depth in the depth of field pass (for very opaque objects) or simply work with the art to minimize the artifacts.

## 5.8 Translucent Shadows


**Figure 16.** *Translucent shadows allow objects such as smoke and explosion to cast shadows.*

Shadow maps are one of the earlier examples of successfully using screen-space information to resolve shadowing problems that otherwise are much more difficult to deal with in world space. In this section, we will show how extending the shadow map's per-pixel information with some extra channels of information can be used to easily augment shadow maps with translucent shadow support.

The shadow map algorithm is extended with a second shadow map that will hold the information for translucent shadows only; the regular shadow map will still contain the information for opaque shadows. In addition, a color buffer to hold the color of the translucent shadows. However, on most hardware a color buffer of the same size of the shadow map needs to be bound whenever the shadow map is rendered, as most hardware does not support having a null color render target. In most games this color buffer is simply set to the lowest bit depth available to minimize the waste of memory, and the color buffer is otherwise not used. We are thus taking advantage of this "free" color buffer.

In the first pass, the opaque shadow map is filled up as it normally would be, rendering only opaque objects. Then, transparent shadow objects are rendered in our second shadow map for transparent objects with z depth write on, no alpha testing and a regular less-equal z-test. This will effectively record the depth of the closest transparent object to the light.

The transparent shadow color buffer is first cleared to white and is then filled by rendering the transparent shadow objects, from front to back, with the render states thay would normally be used in the regular rendering, no depth write and the less-equal z-test. In this stage we will be using the **opaque** shadow map as the z-buffer when filling the color buffer, ensuring no colors are written for translucencies hidden by opaque objects. The front to back ordering is necessary because we are treating these transparencies as light *filters*. The light hits the front transparencies, and gets filtered as it hits each transparent layer in succession.

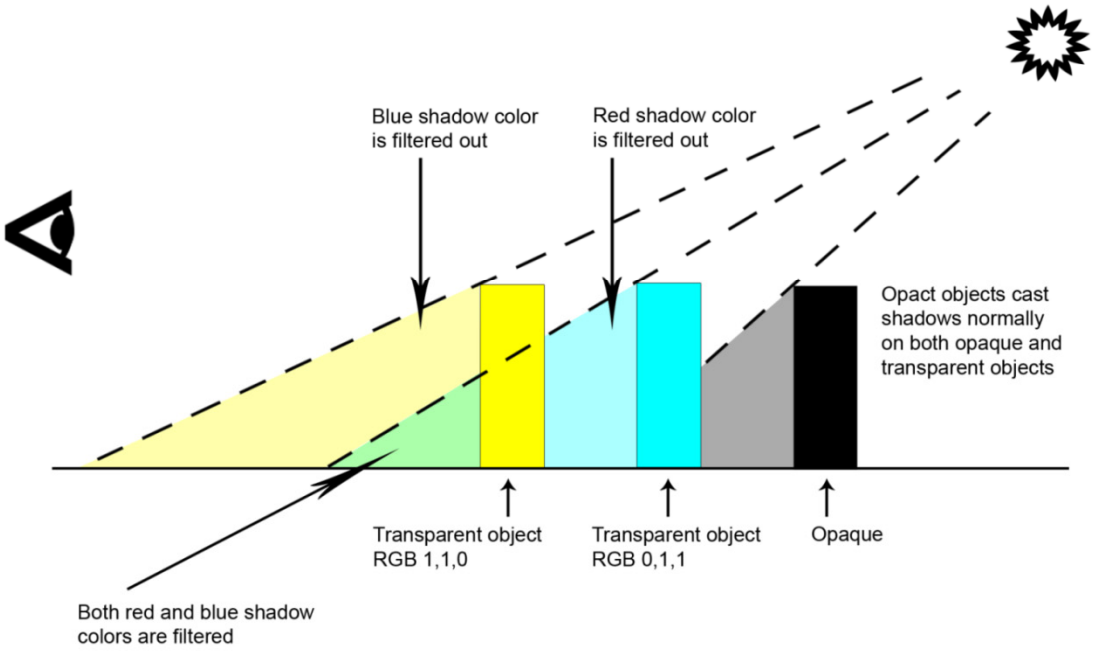


**Figure 17.** *Light filtering process.*

The rendering of the actual translucent shadows then proceeds as follows:

- Perform shadow test with opaque shadow map and translucencies shadow map
- If we failed the translucent shadow map test, modulate by the color in the transparent shadow color map
- Modulate by the result of the opaque shadow map.

This effectively gives us the following desired behavior:

- Passing both tests doesn't color the light;
- Passing the opaque test but failing the translucent test will always color the light by the translucent shadow color;
- Failing the opaque test will always remove the light contribution.

## 5.9 Conclusions

Storage of per-pixel information in off-screen buffers opens up a wide array of possibilities for screen-based effects. Screen-space data sets, although incomplete, are straightforward to work with and can help to unify and simply the rendering pipeline. We've shown here a few ways that we record and use these screen-space data sets in Starcraft II to implement effects such as deferred rendering, screen-space ambient occlusion, depth of field, and translucent shadow maps. The resulting per-pixel techniques are easy to scale and manage and tend to be easier to fit in a constant performance footprint that is less dependent on scene complexity.

## 5.10 References

[CALVER04] CALVER, D. 2004. Deferred Lighting on PS 3.0 with High Dynamic Range, *ShaderX3: Advanced Rendering with DirectX and OpenGL (Shaderx Series)*, Engel, W. (Editor), Charles River Media, Cambridge, MA, November 2004

[SHISHKOVTSOV05], SHISHKOVTSOV, O. 2005. Deferred Shading in S.T.A.L.K.E.R., *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems),* Pharr, M., Fernando, R. (Editors), Addison-Wesley, March 2005.

[KOONE07], KOONE, R. 2007. Deferred Shading in *Tabula Rasa*, GPU Gems 3, Nguyen, H. (Editor), Addison-Wesley, August 2007

[RWPD05] REINHARD, E., WARD, G., PATTANAIK, S. AND DEBEVEC, P. 2005. High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting, Morgan Kaufmann; Har/Dvdr edition, August 2005

[MITTRING07] MITTRING, M. 2007. Finding Next Gen – *CryEngine 2.0*, Chapter 8, SIGGRAPH 2007 Course 28 – Advanced Real-Time Rendering in 3D Graphics and Games, Siggraph 2007, San Diego, CA, August 2007.

[SCHEUERMANNTATARCHUK04] SCHEUERMANN, T. AND TATARCHUK, N. 2004. Improved Depth of Field Rendering, *ShaderX3: Advanced Rendering with DirectX and OpenGL (Shaderx Series)*, Engel, W. (Editor), Charles River Media, Cambridge, MA, November 2004

[DEMERS05], DEMERS, J. 2005. Depth of Field: A Survey of Techniques, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics,* Fernando, R. (Editor), Addison-Wesley, April 2004.