Advanced Real-Time Rendering in 3D Graphics and Games

SIGGRAPH 2006 Course 26

August 1, 2006

Course Organizer: Natalya Tatarchuk, ATI Research, Inc.

Lecturers:

Natalya Tatarchuk, ATI Research, Inc.
Chris Oat, ATI Research, Inc.
Pedro V. Sander, ATI Research, Inc.
Jason L. Mitchell, Valve Software
Carsten Wenzel, Crytek GmbH
Alex Evans, Bluespoon

About This Course

Advances in real-time graphics research and the increasing power of mainstream GPUs has generated an explosion of innovative algorithms suitable for rendering complex virtual worlds at interactive rates. This course will focus on recent innovations in real-time rendering algorithms used in shipping commercial games and high end graphics demos. Many of these techniques are derived from academic work which has been presented at SIGGRAPH in the past and we seek to give back to the SIGGRAPH community by sharing what we have learned while deploying advanced real-time rendering techniques into the mainstream marketplace.

Prerequisites

This course is intended for graphics researchers, game developers and technical directors. Thorough knowledge of 3D image synthesis, computer graphics illumination models, the DirectX and OpenGL API Interface and high level shading languages and C/C++ programming are assumed.

Topics

Examples of practical real-time solutions to complex rendering problems:

- Increasing apparent detail in interactive environments
    - Inverse displacement mapping on the GPU with parallax occlusion mapping
    - Out-of-core rendering of large datasets
- Environmental effects such as volumetric clouds and rain
- Translucent biological materials
- Single scattering illumination and approximations to global illumination
- High dynamic range rendering and post-processing effects in game engines

Suggested Reading

- Real-Time Rendering by Tomas Akenine-Möller, Eric Haines, A.K. Peters, Ltd.; 2nd edition, 2002
- 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.
- ShaderX^3: Advanced Rendering with DirectX and OpenGL, by Wolfgang Engel (Editor), Charles River Media, 1st edition (November 2004)
- ShaderX^4: Advanced Rendering Techniques, by Wolfgang Engel (Editor), Charles River Media, 1st edition (November 2005)
- ShaderX^2: Introductions and Tutorials with DirectX 9.0, by Wolfgang Engel (Editor), Wordware Publishing, Inc.; Book and CD-ROM edition (November 2003)
- ShaderX^2 : Shader Programming Tips and Tricks with DirectX 9.0, by Wolfgang Engel (Editor), Wordware Publishing, Inc.; Book and CD-ROM edition (November 2003)

Lecturers

Natalya Tatarchuk is a staff research engineer in the demo group of ATI's 3D Application Research Group, where she likes to push the GPU boundaries investigating innovative graphics techniques and creating striking interactive renderings. Her recent achievements include leading creation of the state-of-the-art realistic rendering of city environments in ATI demo “ToyShop”. In the past she has been the lead for the tools group at ATI Research. She has published articles in technical book series such as ShaderX and Game Programming Gems, and has presented talks at Siggraph and at Game Developers Conferences worldwide. Natalya holds BA's in Computers Science and Mathematics from Boston University and is currently pursuing a graduate degree in CS with a concentration in Graphics at Harvard University.

Chris Oat is a senior software engineer in the 3D Application Research Group at ATI where he explores novel rendering techniques for real-time 3D graphics applications. As a member of ATI's demo team, Chris focuses on shader development for current and future graphics platforms. He has published several articles in the ShaderX and Game Programming Gems series and has presented at game developer conferences around the world.

Jason L. Mitchell is a software developer at Valve Software, where he works on integrating cutting edge graphics techniques into the popular Half-Life series of games. Prior to joining Valve in 2005, Jason worked at ATI in the 3D Application Research Group for 8 years. He received a BS in Computer Engineering from Case Western Reserve University and an MS in Electrical Engineering from the University of Cincinnati.

Alex Evans started his career in the games industry writing software renderers for innovative UK game developer Bullfrog; after completing a degree at Cambridge University he joined Lionhead Studios full time as one of the lead 3D programmers on the hit game 'Black & White'. His passion is the production of beautiful images through code - both in games, such as Rag Doll Kung Fu and Black & White, but also through his work (under the name 'Bluespoon') creating real-time visuals for musicians such as Aphex Twin, Plaid and the London Sinfonietta.

Carsten Wenzel is a software engineer and member of the R&D staff at Crytek. During the development of FAR CRY he was responsible for performance optimizations on the CryEngine. Currently he is busy working on the next iteration of the engine to keep pushing future PC and next-gen console technology. Prior to joining Crytek he received his M.S. in Computer Science at Ilmenau, University of Technology, Germany in early 2003. Recent contributions include GDC(E) presentations on advanced D3D programming, AMD64 porting and optimization opportunities as well articles in *ShaderX 2*.

Pedro V. Sander is a member of the 3D Application Research Group of ATI Research. He received his Bachelors degree from Stony Brook University, and his Masters and PhD in Computer Science at Harvard University. Dr. Sander has done research in geometric modeling, more specifically efficient rendering techniques and mesh parameterization for high quality texture mapping. At ATI, he is researching real-time rendering methods using current and next generation graphics hardware.

Contents

Preface

Welcome to the Advanced Real-Time Rendering in 3D Graphics and Games course at SIGGRAPH 2006. We've included both 3D Graphics and Games in our course title in order to emphasize the incredible relationship that is quickly growing between the graphics research and the game development communities. Although in the past interactive rendering was synonymous with gross approximations and assumptions, often resulting in 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. Real-time domain is now at the forefront of state-of-the-art graphics research – and who wouldn't want the pleasure of instant visual feedback?

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 highly interactive rates. And as graphics enthusiasts we all strive to produce stunning images which can change in a blink of an eye and let us interact with them. 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 never imagined.

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

Natalya Tatarchuk, ATI Research, Inc.
April, 2006

Chapter 1

Progressive Buffers: View-dependent Geometry and Texture LOD Rendering

Pedro V. Sander[1]
ATI Research

Jason L. Mitchell[2]
Valve Software

The content of this chapter also appears on Symposium on Geometry Processing 2006

1.1 Abstract

We introduce a view-dependent level of detail rendering system designed with modern GPU architectures in mind. Our approach keeps the data in static buffers and geomorphs between different LODs using per-vertex weights for seamless transition. Our method is the first out-of-core system to support texture mapping, including a mechanism for texture LOD. This approach completely avoids LOD pops and boundary cracks while gracefully adapting to a specified frame rate or level of detail. Our method is suitable for all classes of GPUs that provide basic vertex shader programmability, and is applicable for both out-of-core or instanced geometry. The contributions of our work include a preprocessing and rendering system for view-dependent LOD rendering by geomorphing static buffers using per-vertex weights, a vertex buffer tree to minimize the number of API draw calls when rendering coarse-level geometry, and automatic methods for efficient, transparent LOD control.

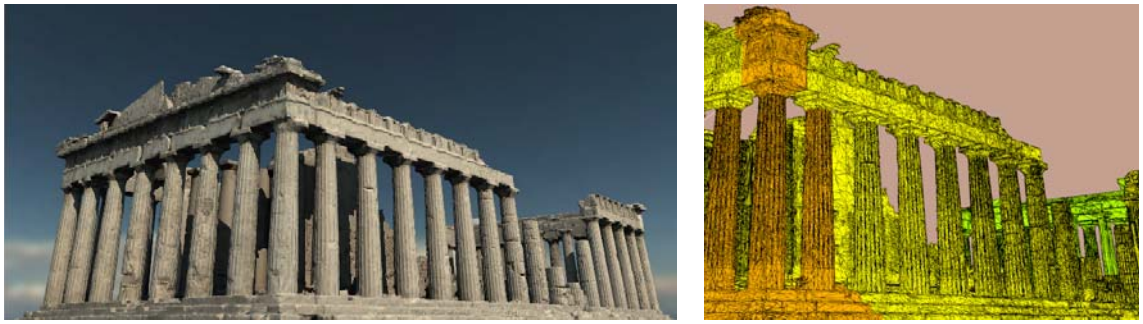
1.2 Introduction

Real-time rendering of massive 3D scenes lies at the forefront of graphics research. In this paper we present new algorithm for real-time rendering of large polygonal meshes. To our knowledge, this is the first out-of-core view-dependent mesh renderer that supports texture mapping and continuous smooth transitions between LODs to prevent

[1] psander@ati.com
[2] jasonm@valvesoftware.com

popping. Both of the above features allow our method to faithfully render geometry with high fidelity without requiring sub-pixel sized triangles with Gouraud-interpolated vertex colors. Our method is also applicable to instanced geometry, as we will show in the results section.

Our data structure, the progressive buffer (*PB*), is derived from a progressive mesh (*PM*) [Hop96] and consists of a sequence of static buffers at different levels of detail for the different clusters of polygons that make up the mesh. Each buffer stores an irregular mesh, thus faithfully capturing geometric detail for a given polygon rate. Transitioning between different levels of detail is achieved via geomorphing [Hop96]. Our novel method computes geomorphing weights per vertex in order to ensure consistency between neighboring clusters of triangles and to prevent boundary discontinuities. Figure 1 shows a rendering of a *PB* with a texture and with color-coded LODs.



(a) Texture-mapped rendering with shadows *(b) Color-coded chart LOD*

Figure 1. *View-dependent geometry and texture LOD on a 16M triangle mesh. The adaptive model being rendered has 800,000 triangles. This scene is rendered at 30fps.*

Due to the usage of static buffers and texture mapping, this system achieves high rendering rates using consumer graphics hardware and scales to previous hardware.

This paper presents a preprocessing method and a rendering system for geometry and texture view-dependent dynamic level of detail that is suitable for a large class of graphics hardware. In order to achieve this objective, we introduce the following techniques:

- A rendering method that geomorphs the geometry in the vertex shader using per-vertex weights. This approach completely prevents LOD pops and boundary cracks, while still using "GPU-friendly" static vertex and index buffers.

- A hierarchical method to more efficiently render geometry that is far from the viewer, thereby reducing the number of API draw calls.

- A scheduling algorithm to load required geometry and texture data on demand from disk to main memory and from main memory to video memory.

- An automatic method that controls and smoothly adjusts the level of detail in order to maintain a desired frame rate. This approach is transparent and gracefully adapts the rendering quality as a function of the graphics horsepower and the scene's geometric complexity.

The approach presented in this paper has the following drawbacks and limitations:

- On current hardware, the size of the vertex buffer is doubled when geomorphing to a lower level of detail. Note, however, that this secondary buffer only needs to be loaded when a particular cluster is in a geomorphing region (see Section 3). Since, high-detail geometry is only required for regions that are close to the camera, the benefit of a flexible data structure outweighs the overhead on the small subset of buffers that reside in video memory.

- Our method requires a larger number of draw calls than purely hierarchical algorithms. This is required because current graphics hardware does not allow changing texture state within a draw call. Grouping separate textures in unified atlases at higher levels of detail would change the texture coordinates, thus preventing those from being geomorphed appropriately. We believe the advantages of texture mapping are more important than the efficiency gain of having fewer draw calls on clusters near the camera. For clusters far from the camera, we address this problem by grouping the low resolution data in unified buffers, thus reducing the number of draw calls on large scenes, where it matters the most.

- Although this approach does not require a perfect voxelization of space to construct different clusters of adjacent faces, our rendering method achieves better results when there are no clusters significantly larger than the average. This is because the maximum cluster radius restricts the size of the LOD regions as described in Section 4. For best performance, clusters should have similar bounding radii (within each connected component). We address this by first voxelizing space, and then further splitting each cluster into charts that are homeomorphic to discs and thus can be parametrized.

The remainder of this paper is organized as follows. In Section 2, we describe previous work and how it relates to our approach. Section 3 outlines our basic data structure, the progressive buffer, which provides a continuous level of detail representation for the mesh. In Section 4, we describe how we efficiently render progressive buffers. Section 5 presents our preprocessing algorithm, which partitions the mesh into clusters and generates the progressive buffers for each cluster. Finally, we present results in Section 6 and summarize in Section 7.

1.3 Previous Work

Several methods for efficient rendering of large polygon models have been proposed in the past. The earlier works focused on continuous LOD, while more recent research addresses rendering large models that do not fit in video memory, thus opening a number of different issues, such as out-of-core simplification and memory management.

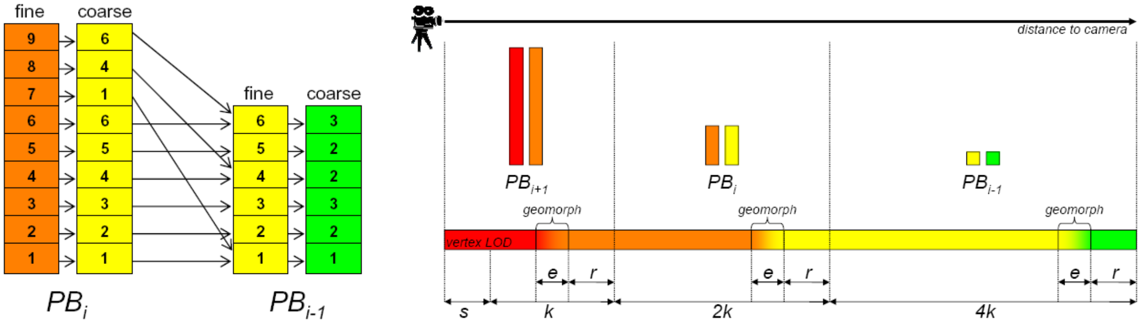
The first approaches developed for view-dependent real time mesh rendering adaptively simplified at the triangle level via edge collapses [Xia96, Hoppe97, El-Sana99]. With the advent of programmable graphics hardware it has become much more efficient to perform larger scale simplification on static buffers. Other methods clustered sets of vertices in a hierarchical fashion [Luebke97]. While these methods are generally good at providing view dependent LOD, none of the above methods are applicable to out-of-core rendering of arbitrary polygonal meshes.

An alternative approach for rendering large meshes was presented by Rusinkiewicz and Levoy [Risinkiewicz00]. Their method converts the input mesh to a vertex tree, which is then rendered using point primitives. However, current graphics hardware is more optimized for rendering triangle primitives with texture/normal maps, which usually produces higher quality results for the same rendering cost. There are several hybrid approaches that use both triangle and point primitives in order to reduce rendering cost (e.g., [Chen01, Dey02]).

Recent out-of-core methods for view-dependent mesh rendering have focused on the fact that graphics hardware is significantly more efficient when rendering static triangle buffers from video memory. These methods use irregular meshes, the most common used path on current graphics cards, making them very efficient for a given fidelity. There are several recent methods for out-of-core rendering based on irregular meshes (e.g., [El-Sana00, Vadrahan02, Lindstrom03, Cignoni04, Yoon04]). These methods address issues of memory management and prefetching to video memory. However, to our knowledge, none of the previously published works for out-of-core rendering of arbitrary polygonal meshes provide a continuous smooth transition between different LODs nor do they support texture mapping. The above methods rely on the fact that, with a very high triangle throughput rate, one can store the required attribute data per vertex and directly switch the rendered triangles to a coarser level of detail before the change becomes visually noticeable (i.e., before a screen space error tolerance is met).

Our novel approach geomorphs between the levels of detail, resulting in a smooth *pop-free* transition, regardless of the screen-space error of the approximation. It does not require pixel-sized triangles, as it can store detail in texture maps and provide LOD control over the texture images. Therefore, since the rendered geometry can be coarser, it allows us to focus the GPU resources on rendering other scene effects with complex shaders while still maintaining real-time frame rates. Furthermore, our method can be used with older generation graphics hardware for a given loss of rendering quality.

Gain and Southern [Gain03] use geomorphing for static LOD within each object of the scene. Our method, however, addresses multiple levels of detail for a single arbitrary object, thus allowing for view-dependent LOD of large meshes. This is accomplished by computing the geomorphing weight per vertex, as opposed to per object, by properly constructing mesh clusters, and by constraining where LOD transitions take place, as described in the next section. [Ulrich02] presents a method designed for terrain rendering and avoids transitions between objects of mismatched LODs by introducing a small vertical ribbon mesh, which would likely produce more significant texture-stretching artifacts for arbitrarily complex geometry. Our approach addresses arbitrary meshes and avoids the need for ribbon meshes by performing the geomorph computation per vertex. The idea of per-vertex LOD was first introduced by the multiresolution rendering algorithm of Grabner in 2003.



(a) Transition between LODs.

(b) Vertex LOD as a function of distance from camera.

Figure 2. *The geomorph depicted on the left occurs when the cluster's bounding sphere's center is near the transition point between rendering PB_i and PB_{i-1}. To ensure continuity, the geomorph must be performed at a distance of r away from this transition point, where r is the maximum cluster radius. This is necessary so that all vertices have finished geomorphing when the cluster switches LOD. The variables k, s, and e determine the size of the LOD bands and the geomorph region.*

1.4 The progressive buffer

As mentioned above, our approach consists of constructing a progressive buffer, which is a series of vertex and index buffers that represent a mesh at different levels of detail (LODs). Figure 2a shows two levels of detail of a progressive buffer: PB_i and PB_{i-1}. Note that there are two vertex buffers associated with each level of detail. Each cell represents a vertex, which is identified by an ID number. Each level of detail also has an index buffer, which is omitted from the figure. We will refer to the set of two vertex buffers and one index buffer at a particular level of detail as a static buffer (PB_i, where i is the level of detail), and to the entire sequence of static buffers as a progressive buffer.

Continuous level of detail. The discrete static buffers, coupled with continuous geomorphing weights yield a continuous level of detail representation for the mesh. When rendering a static buffer, geomorphing weights are determined in order to properly blend the vertex data between the fine and coarse buffers based on distance from the camera. The coarse buffer of PB_i contains the same vertex data as the fine buffer of PB_{i-1}. Figure 2b shows a progressive buffer with 3 levels of detail. Note that, as the distance from the camera increases, the cluster is geomorphed to the coarser buffer and subsequently switches to a different static buffer. As long as the geomorph to the coarser buffer is completed before the switch, there will be no difference in the rendered image when switching from one static buffer to the next.

View-dependent level of detail. So far, this method works well for static level of detail, where the entire mesh is rendered using the same static buffer and geomorphing weight. However, in order to enable view-dependent dynamic level of detail, we must be able to assign different levels of detail to different regions of the mesh. To achieve this, we partition the mesh into multiple clusters and construct a progressive buffer for each cluster. In order to prevent geometric cracks on cluster boundaries, we must meet the following requirements:

- When constructing the progressive buffers, consistently simplify all clusters of each connected component in unison in order to achieve consistent cluster boundary vertex positions at all LODs, as described in Section 5.

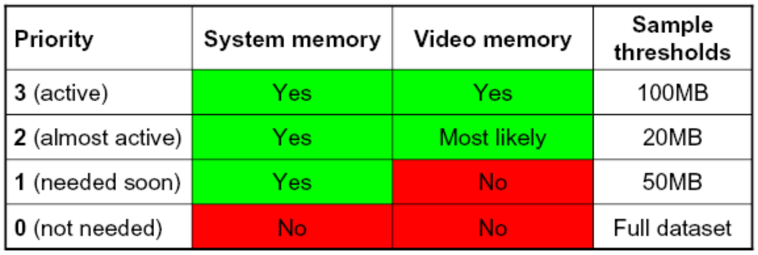
- Ensure that the LOD and geomorphing weights of boundary vertices match exactly across clusters, as described next.

Clearly, one cannot assign a constant LOD for the entire cluster; otherwise all clusters of a connected component would need to have the same LOD for all boundaries to match. That would not allow for dynamic level of detail. To address this issue, we compute the geomorph weights per vertex. If the geomorph weight is determined based on the distance from the vertex to the camera, a boundary vertex will have the same LOD and geomorph weight as its mate on the neighboring cluster. This approach avoids boundary cracks and allows the level of detail to vary across the mesh. Note that the discrete static buffer is constant through the entire cluster. It is determined based on the distance from the cluster's bounding sphere center to the camera.

The vertex LOD bar in Figure 2b shows that as long as the proper buffers are used, one can render a cluster by geomorphing each vertex independently, based on its distance to the camera. The distance range in which the geomorph takes place must be at least *r* away from the LOD boundary, where *r* is the maximum cluster bounding sphere radius of the mesh. This is necessary in order to ensure that none of the vertices will be in the geomorph range after the cluster's bounding sphere center crosses the discrete LOD boundary and the renderer starts using a different static buffer for that cluster. As shown in Figure 2b, we choose the geomorph range to be as far away from the camera as possible in order to maximize the quality of the rendering.

Coarse buffer hierarchy (CBH). In order to minimize the number of draw calls, we group the static buffer of the coarsest LOD of all clusters in a single vertex buffer with a corresponding index buffer. We then render different ranges of this buffer with the aid of a hierarchical data structure which groups clusters together. This approach, detailed in Section 4.3, also allows us to perform frustum culling at any node of the tree.

Out of core data management. During rendering of an out of core model, the engine keeps track of the continuous LOD determined by the center of the bounding sphere of each cluster. As this number changes, the engine appropriately loads and unloads data to and from the disk, main memory, and video memory. We employ a system that has four priority levels, as shown in Figure 3. Active buffers that are currently being rendered must reside in video memory and have priority 3. Buffers that could become active very shortly if the distance from the camera to the cluster changes slightly have priority 2 and are also loaded to video memory (this buffer prefetching is very important to ensure the buffer is available when needed). Buffers that could possibly be needed in the near future have priority 1 and are loaded to main memory, but not to video memory. Finally, all other buffers have priority 0 and only reside on disk. A least-recently-used (LRU) scheme is used to break ties between buffers that have the same priority level. As shown in Figure 3, the engine can set thresholds to each of these priority levels based on the amount of video and main memory present and how fast it can read from the hard disk. Methods to automatically adjust the complexity of the scene given fixed memory thresholds or current rendering frame rate are described in Section 4.4.



Priority	System memory	Video memory	Sample thresholds
3 (active)	Yes	Yes	100MB
2 (almost active)	Yes	Most likely	20MB
1 (needed soon)	Yes	No	50MB
0 (not needed)	No	No	Full dataset

***Figure** 3. Different priority levels along with where the buffers reside and example maximum thresholds.*

Texture mapping. Progressive buffers can be texture mapped using a consistent mesh parametrization. [Cohen98] described an approach to preserve texture coordinates during simplification. This method extends naturally to progressive buffers. A single texture can be used for the entire progressive buffer. Each mip level of the texture is associated with a static buffer. Thus, the higher the static buffer being used, the higher the maximum mip level. As with the geometry data, texture data is also stored on disk and loaded out of core as the level of detail changes.

1.5 Rendering

In this section, we describe how to efficiently render progressive buffers. We first describe a basic algorithm using the data structure described in the previous section. Then we describe an optimized hierarchical algorithm to reduce the number of draw calls. Finally, we describe how to adjust the level of detail to maintain a stable frame rate.

1.5.1 Computing the level of detail

In order to render the mesh, our rendering algorithm must determine in real-time which level of detail we want to use for each cluster. Our approach determines the level of detail based on the cluster's distance to the camera and tries to maintain a constant triangle size after projection to the screen. Assuming the worst case scenario, in which the triangles in the cluster are all facing the viewer straight-on, this method maintains an approximately constant screen-space area for the triangle as the camera moves. As the distance to the camera doubles, the screen space area of the triangle is reduced by a factor of four. As a result, every time the distance to the camera doubles, we switch to the next coarser level of detail, which has four times fewer vertices. Note that, as shown in Figure 2, this is only true if the parameter *s* is set to its default value of 0. The variable s, however, can be set to a positive or negative value in order to further adjust the LOD. One can consider other distance and vertex ratios, but one significant advantage of each LOD having four times more vertices than its parent is that the same factor of four can be applied to the textures, which is convenient, especially when mipmapping these textures. This way, both vertex and texture detail change by the same factor from one LOD to the next.

The variables *s* and *k* from Figure 2 can be adjusted as a function of several values, such as frame rate, memory and triangle count upper bound. *s* is used as a bias term for the LOD, while *k* is a scaling term. Section 4.4 describes how to automatically adjust these values to maintain a given frame rate.

We set the variable *e*, which represents the length of the geomorph band, to its maximum allowed value of *k* − *r*. This makes the transitions smoother and does not affect rendering performance since the GPU still processes the same number of triangles.

Given *s*, *k* and *d*, which is the distance from the cluster's center to the camera, the level of detail of a cluster is

$$i = floor\left(\log_2\left(\frac{d-s}{k}+1\right)\right)$$

Prior to rendering the cluster, we must also determine the start distance, d_s, and the end distance, d_e for the geomorph region within that cluster, which is computed as follows:

$$d_e = \left(2^{i+1}-1\right)k + s - r$$

$$d_s = d_e - e$$

These two values must be placed in the GPU's constant store, so that during rendering, the vertex shader can interpolate the position and other attributes based on the vertex distance from the camera.

1.5.2 Basic rendering algorithm

The basic rendering algorithm traverses all of the clusters and, for each cluster, tests the cluster's bounding sphere against the view frustum. Should the cluster be inside the frustum, it then sets the appropriate constant store variables and renders the desired level of detail. The buffers representing that level of detail should already reside on the graphics card, due to their high priority, unless the amount of available graphics memory is not sufficient to render the scene. The following vertex shader pseudo code properly geomorphs between the two positions, p_1 and p_2:

```
d = length(p1 - eye);
w = smoothstep(ds, de, d);
Pos = lerp(p1, p2, w);
```

Normals and texture coordinates are also geomorphed this way. Texture coordinates can be morphed because a consistent parametrization is generated for all LODs. Note that normals need to be renormalized after geomorphing.

The pixel shader performs two texture fetches from the two textures (corresponding to the two LODs), and then interpolates between them using the same interpolation weight w.

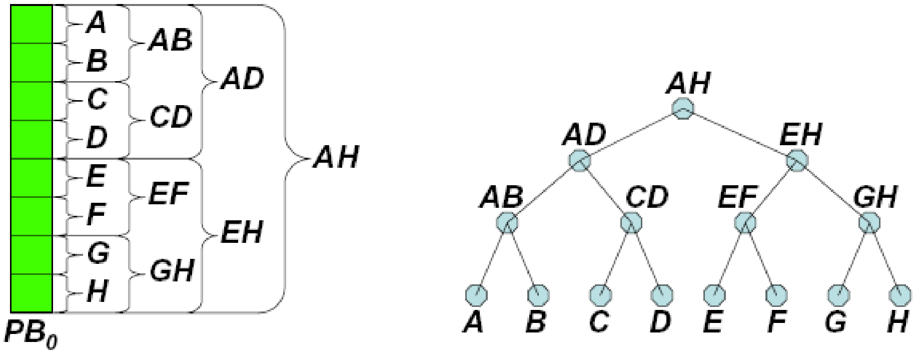


***Figure 4**. The coarse buffer for the entire mesh (left) and its accompanying hierarchy, which is used to minimize the number of API draw calls.*

1.5.3 CBH rendering algorithm

In order to reduce the number of draw calls, group clusters at the coarsest LOD level. Since the coarsest buffers take little space, they can always be loaded into video memory.

As shown in Figure 4, the coarse buffers are grouped in an order dictated by the coarse buffer hierarchy (CBH). The CBH is a tree that contains all the clusters of the mesh as leaves. When rendering, the engine parses this hierarchy, and when it finds that all clusters below a certain node in the tree are in the coarsest level (by testing the node's bounding sphere), it renders all those clusters using a single draw call. For instance, node CD in the tree can render clusters C and D with a single draw call, since they are adjacent in the index buffer. The node simply stores the starting index and the number of triangles. The coarsest buffer does not contain parent information, since there are no coarser LODs.

In order to take advantage of early-Z culling hardware, we first render all nodes that are not in the coarsest level, since they represent the front-most geometry. We then render all coarse nodes by performing a depth-first CBH traversal. First, all leaves that are at the coarsest level are tagged "renderable." Then, the recursive algorithm first traverses the children of a particular node and, if they are both tagged renderable, the current node is tagged as renderable. Otherwise, if one of the children is tagged as renderable, it adds that child node to the list of nodes to be rendered. After the tree traversal is complete, all nodes in the list are rendered using a smaller number of API draw calls.

1.5.4 Level of detail control

In this section, we describe how we can automatically control the level of detail. This allows our method to work at satisfactory frame rates on a wide variety of GPUs.

The level of detail is adjusted by increasing and decreasing the *k* variable from Figure 2b. When this variable reaches its minimum value or $r + e_{min}$, the *s* variable can be decreased to a value smaller than 0 if the level of detail must be further decreased.

What remains to be determined is whether, given the current runtime state, we want to increase or decrease the level of detail. Ideally, we would simply use the frame rate to determine whether we want to increase or decrease the level of detail. For instance, if it is above 65 fps, we increase the LOD, and if it is below 55 fps we decrease the LOD. However, oftentimes frame rate only changes after an undesirable event has already taken place. For instance, if the active textures exceed the capacity of video memory, the frame rate suddenly drops. Ideally, we would like to prevent such drops. In order to achieve this, we propose setting a video memory upper bound, a system memory upper bound, a triangle-count upper bound, and a frame rate lower bound. The level of detail is constantly and slowly increased unless one of these bounds is violated, in which case, it decreases. Using these bounds on the best indicators of performance, we prevent such drastic frame rate changes. Naturally, the bounds can be tuned to the system configuration. Preprocessing is not affected by these changes and the same progressive buffer data structures can be used on all systems.

1.6 Preprocessing algorithm

In this section, we describe the steps involved in converting a triangle mesh into a progressive buffer. These preprocessing stages are mainly based on previous techniques for simplification and texture mapping of LOD representations.

1.6.1 Segmentation

To segment an input mesh into clusters, we use a voxelization of space. This ensures that the bounding spheres of all the clusters are bound based on the size of the voxels. This, however, may result in voxels that parameterize with high distortion, have annuli, and are composed of disconnected components. To address these problems, we further split the clusters into charts that are mostly planar and homeomorphic to discs. To achieve this, one can do this chartification manually or use one of several existing chartification algorithms (e.g., [Maillot93, Levy02, Sander03]). These charts can then be parametrized and their attributes can be stored in a single texture atlas for each cluster.

1.6.2 Hierarchy construction

To build the CBH, we start with all of the clusters as leaves of the tree and then perform a bottom-up greedy merge. All possible merging pairs are placed in a priority queue, sorted by smallest bounding sphere of the resulting merged cluster.

The above approach ensures that nearby clusters will be grouped together. Disconnected components can also be merged together, as long as they are using the same LOD band sizes (i.e., the same *k*, *s*, *e*, and *r*).

1.6.3 LOD-compliant parameterization

In order to texture map the mesh, each chart must parametrized into a disc and packed into atlases for each cluster. The parametrization restrictions for chart boundaries are discussed in [Cohen98] and [Sander01]. Any parametrization metric can be used (e.g., [Floater97, Sander01, and Desbrun02]). For our examples, we used the L^2 stretch metric from [Sander01], which minimizes sampling distortion from the 3D surface to the 2D domain.

The computed parametrization and resulting texture mip map will be applicable to all levels of detail. In the next section, we will describe how we simplify the mesh to guarantee this.

1.6.4 Progressive mesh creation

In this step, we must simplify the mesh, ensuring that the edge collapses keep the chart boundaries consistent and do not cause flips in UV space [Cohen98]. We use the half-edge collapse with the memory-less appearance preserving simplification metric as in [Sander01].

Each connected component must be simplified in unison. In order to achieve this, we simplify one cluster at a time until it reaches a specified user defined geometric error threshold. The order in which we simplify the clusters does not significantly affect the results, since we perform the same number of simplification passes on all the clusters between each pair of adjacent levels of detail. In order to consistently simplify the cluster boundaries, when simplifying each cluster, we also load the adjacent clusters and simplify the boundary vertices as well. However, we keep the neighboring cluster’s interior vertices fixed. This allows all boundaries to be simplified and prevents boundary cracks. This method is related to that presented by [Hoppe98] for terrain simplification and by [Prince00] for arbitrary meshes.

1.6.5 Vertex and index buffer creation

Now that a *PM* has been constructed, we must extract meshes at different levels of detail and create corresponding vertex and index buffers. As mentioned previously, we chose each level of detail to have four times fewer vertices than the next finer one. For our examples, we picked five levels of detail, as that resulted in a sufficient range of LODs.

After extracting the meshes at the different LODs, we construct a set of vertex and index buffers for each cluster of each LOD. Each vertex will not only contain its own attributes (e.g., position, normal, texture coordinates), but it will also contain all the attributes of its parent vertex in the next coarser LOD. The *PM* hierarchy provides the ancestry tree. The parent vertex is its closest ancestor that is of a coarser level of detail. If the vertex is in the next coarser level, the parent is itself.

After these buffers are created, the vertices and faces are reordered using the method of [Hoppe99] to increase vertex cache coherency and thus improve rendering performance.

Vertex buffer compression. One limitation of this work is the doubled vertex buffer size that is required to render progressive buffers on current graphics architectures. In an attempt to offset this, we store the buffers with 28 bytes per vertex. Each of the two normals is stored with 10 bits per component for a total of 4 bytes per normal. Each set of 2D texture coordinates is stored using two 16-bit integers, thus occupying 4 bytes each. Finally, each of the two sets of 3D position coordinates is stored using three 16-bit integers each, for a total of 6 bytes each.

While the precision for the normals and texture coordinates is sufficient, 16-bit precision for the position in a large scene is not high enough. In order to address this, we initially considered storing the position as a delta from the chart's center position. However, this would result in dropped pixels at chart boundaries because of precision errors when decompressing vertices from adjacent charts whose positions must match exactly. In order to avoid this problem, we store the position modulo *r* (*p* % *r*), where *r* is the largest cluster bounding sphere. Given the stored position and the cluster's bounding sphere center position (which is made available through the constant store), the vertex shader can reconstruct the original position. Since the stored values (*p* % *r*) are identical for matching vertices on adjacent clusters, they are reconstructed to the exact same value.

1.6.6 Texture sampling

Next, we sample the texture images. In our cases, the textures were filled using data from other texture maps that use a different set of texture coordinates, or the textures were filled with normals from the original geometry. Next, we fill in samples near the boundaries using a dilation algorithm to prevent mipmapping and bilinear sampling artifacts. We then compute mipmaps for the textures. Each level of the mipmap corresponds to the texture that will be used as the highest mip level of a particular LOD.

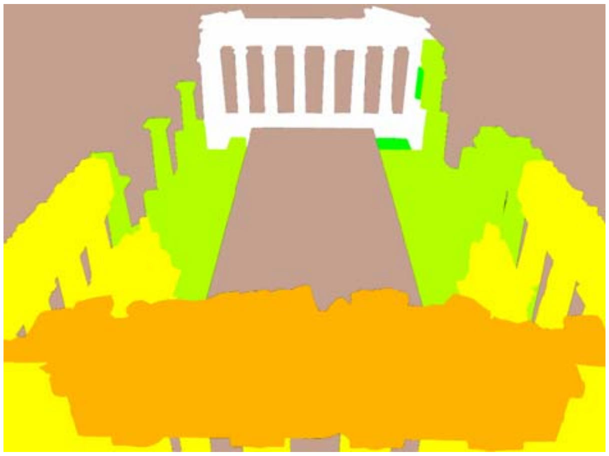


Figure 5. *Visualization of the levels of detail.*

1.7 Implementation and results

We implemented progressive buffers in DirectX 9.0. Our experiments were made on a Pentium 4 2.5GHz machine with 1GB of memory and a Radeon X800 graphics board.

In order to analyze our algorithm, we preprocessed the Pillars model from Figure 7, a 14.4 million polygon textured model with 288 voxels. We used an input model that was split into parametrizable charts. The remaining steps were performed automatically as outlined in Section 5. Each voxel used a 512x512 texture image at the highest level of detail.

Figure 6 shows results for texture-mapped rendering with two different configurations for the Pillar model. The top row of graphs shows statistics using a fixed LOD band size. Note that the frame rate remains at about 60fps until the end of the fly path, when the camera starts moving away from the model. At that point, the frame rate increases, while memory usage, number of faces rendered and draw calls drop. The bottom row shows statistics for the same fly path, except that instead of using a fixed LOD band size, it tries to maximize the band size subject to a target of 60MB of video memory usage. As evidenced by the second chart, memory remains roughly constant around 60-70MB, causing the LOD band sizes to shrink and grow automatically to meet that memory requirement. This is a significantly lower memory footprint than on the first experiment, and therefore the number of rendered faces decreases by a factor of two and the frame rate increases to approximately 90fps.

The vertex caching optimization gave an improvement of almost a factor of three in rendering speed. Peak rates for our system were in the 60Mtri/sec range, which we consider high given that we are decompressing and geomorphing between two buffers in the vertex shader.

Figure 5 shows the different LODs. The lowest level of detail is shown in dark green. Coarse LODs, whose draw calls are grouped together using the CBH, are shown in white. Figure 7 shows the LODs of the Pillars model from different vantage point. Note that the view point is close to the right-most pillar.

Figure 8 illustrates the importance of prefetching by graphing the number of clusters that were unavailable for rendering when prefetching from disk was disabled (no distinction between priority levels 0, 1 and 2).With prefetching enabled, and loading approximately 30% additional buffer data, all clusters are available.

Figure 1 shows an example of shadow-mapping on our system. A 16M triangle Parthenon mesh was used for this example. Shadows were cast with the coarse geometry by rendering the coarse mesh to the shadow map with just a single API draw call. That scene was rendered at 30fps.

Figure 9 shows examples of using our system for instancing. In this case, the progressive buffer is loaded into memory and instanced multiple times. The total number of virtual triangles is 45 million for the planes scene, and 240 million for the dragon scene. However, less than 1 million triangles are actually rendered when using the LOD system.

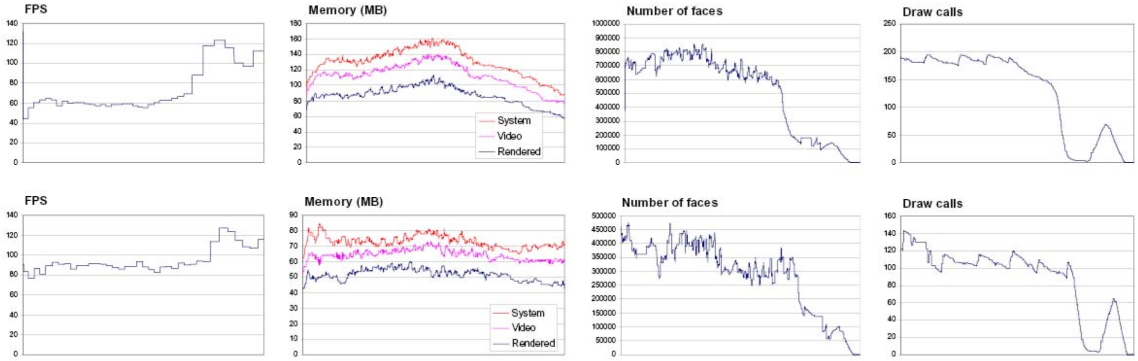


Figure 6. *Results for texture-mapped rendering. The results of using a fixed LOD band size, k, is shown in the top four graphs. The bottom four graphs show the results of automatic LOD control with a target of 60MB of video memory usage.*

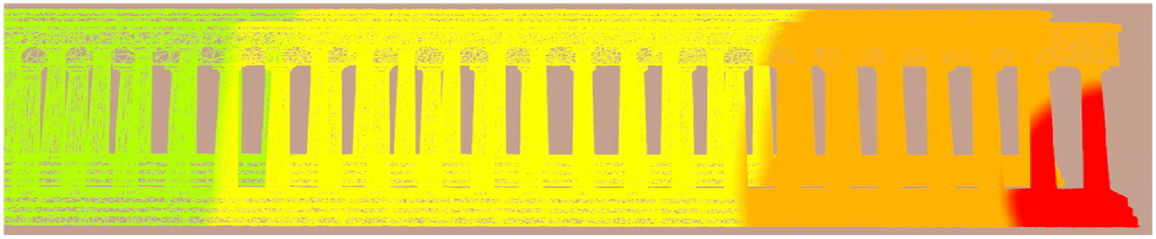


Figure 7. *Wireframe rendering of the color-coded LODs for the Pillars model. Note the significantly lower tessellation to the left, where it is far from the camera.*

1.8 Summary and future work

We presented a new data structure and algorithm for dynamic level of detail rendering of arbitrary meshes. We showed examples with out-of-core and instanced geometry. To our knowledge, our out-of-core view-dependent renderer is the first such system to provide smooth LOD transitions and texture mapping, the latter being a key component of real-time graphics. We presented experiments that demonstrate the viability of such a geometry and texture LOD approach. The method allows for scales well, is suitable for current and previous graphics hardware.

There are several interesting areas of future work:

Deformable models. Although we have not implemented progressive buffers for deformable models, our approach can be adapted to such a setting. The renderer would need to be able to track a bounding sphere for each model, and be aware of the maximum radius *r* of all clusters over all of their possible poses (that is necessary in order to set the geomorph range as shown in Figure 2b). All of these quantities can be preprocessed. The bounding spheres would need to be propagated up the CBH when they change.

Fly path lookahead. If the camera follows a specific known path, such as in a presentation, an architectural walkthrough or a demo, the prefetching algorithm can be adapted based on this fly path, since the application can easily determine which clusters will be needed ahead of time.

Tiled geometry. This technique can also be applied to a streaming world system, commonly used in video games. Only a single copy of each tile needs to be stored in video memory. The different tiles would have to be simplified in a consistent way, so that the vertices would match at the boundaries at all LODs.

Future architectures. In future graphics architectures, with performant texture fetch in the vertex shader, one could consider storing the parent index rather than the parent vertex attributes, thus reducing memory overhead.

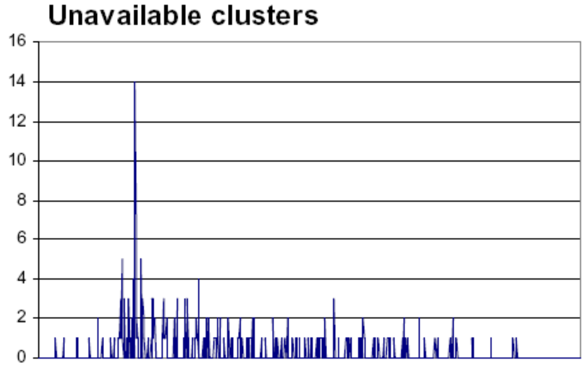
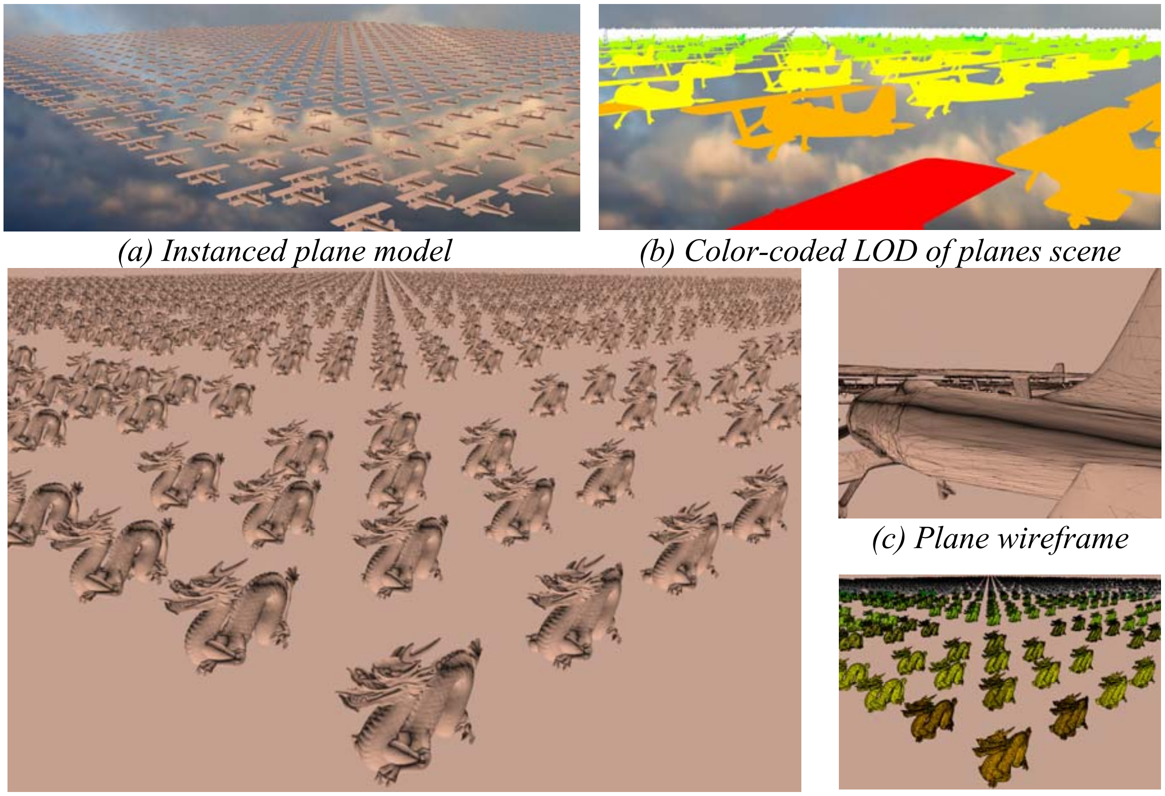


***Figure 8**. Number of unavailable cluster buffers when following a fly path with prefetching disabled. If prefetching is enabled, there were no unavailable buffers for the same fly path.*



(a) Instanced plane model

(b) Color-coded LOD of planes scene

(c) Plane wireframe

(d) Instanced dragon model

(e) LOD of (d)

***Figure 9**. Examples of instancing: 900 planes for a total of 45 million triangles and 1600 dragons for a total of 240 million triangles.*

1.9 Acknowledgements

We would like to thank Eli Turner for his help with the artwork. We thank Toshiaki Tsuji for the I/O and thread management code from his library, as well as help with optimizing the code. Finally, we thank Thorsten Scheuermann, John Isidoro and Chris Brennan for interesting discussions about the algorithm, and the reviewers for their comments and suggestions.

1.10 Bibliography

CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.. 2004. Adaptive TetraPuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. In Proc. SIGGRAPH ()

CHEN B., NGUYEN M. 2001. POP: A hybrid point and polygon rendering system for large data. In Visualization 2001

COHEN J., OLANO M., MANOCHA D. 1998. Appearance-preserving simplification. In Proc. SIGGRAPH

DEY T., HUDSON J.: PMR. 2002. Point to mesh rendering, a feature-based approach. In Visualization 2002

DESBRUN M., MEYER M., ALLIEZ P. 2002. Intrinsic parameterizations of surface meshes. Computer Graphics Forum 21, 209–218.

EL-SANA J., CHIANG Y.-J. 2000. External memory view-dependent simplification. Computer Graphics Forum 19, 3, 139–150.

EL-SANA J., VARSHNEY A. 1999. Generalized view-dependent simplification. Computer Graphics Forum 18, 3, 83–94.

FLOATER M. S. 1997. Parametrization and smooth approximation of surface triangulations. Computer Aided Geometric Design 14, 231–250.

GAIN J., SOUTHERN R. 2003. Creation and control of real-time continuous level of detail on programmable graphics hardware. Computer Graphics Forum, March

HOPPE H 1996. Progressive meshes. In Proc. SIGGRAPH

HOPPE H. 1997. View-dependent refinement of progressive meshes. In Proc. SIGGRAPH

HOPPE H. 1998. Smooth view-dependent level-of-detail control and its applications to terrain rendering. In Visualization 1998

HOPPE H. 1999. Optimization of mesh locality for transparent vertex caching. In Proc. SIGGRAPH

LUEBKE D., ERIKSON C. 1997. View-dependent simplification of arbitrary polygonal environments. In Proc. SIGGRAPH

LINDSTROM P. 2003. Out-of-core construction and visualization of multiresolution surfaces. In ACM 2003 Symposium on Interactive 3D Graphics.

LÉVY B., PETITJEAN S., RAY N., MAILLOT J. 2002. Least squares conformal maps for automatic texture atlas generation. In Proc. SIGGRAPH

MAILLOT J., YAHIA H., VERROUST A. 1993. Interactive texture mapping. In Proc. SIGGRAPH

PRINCE C. 2000. Progressive Meshes for Large Models of Arbitrary Topology. Master's thesis, Department of Computer Science and Engineering, University of Washington, Seattle

RUSINKIEWICZ S., LEVOY M. 2000. QSplat: A multiresolution point rendering system for large meshes. In Proc. SIGGRAPH

SANDER P. V., SNYDER J., GORTLER S., HOPPE H. 2001. Texture mapping progressive meshes. In Proc. SIGGRAPH

SANDER P. V., WOOD Z. J., GORTLER S. J., SNYDER J., HOPPE H. 2003. Multi-chart geometry images. In Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry Processing

ULRICH T. 2002. Rendering massive terrains using chunked level of detail control. SIGGRAPH 2002 Course Notes.

VADRAHAN G., MANOCHA D. 2002. Out-of-core rendering of massive geometric environments. In Visualization 2002

XIA J., VARSHNEY A. 1996. Dynamic view-dependent simplification for polygonal models. In IEEE Visualization '96 Proceedings

YOON S.-E., SALOMON B., GAYLE R., MANOCHA D. 2004. Quick-VDR: Interactive view-dependent rendering of massive models. In Visualization 2004.

Chapter 2

Animated Skybox Rendering and Lighting Techniques

John R. Isidoro[3] and Pedro V. Sander[4]
ATI Research

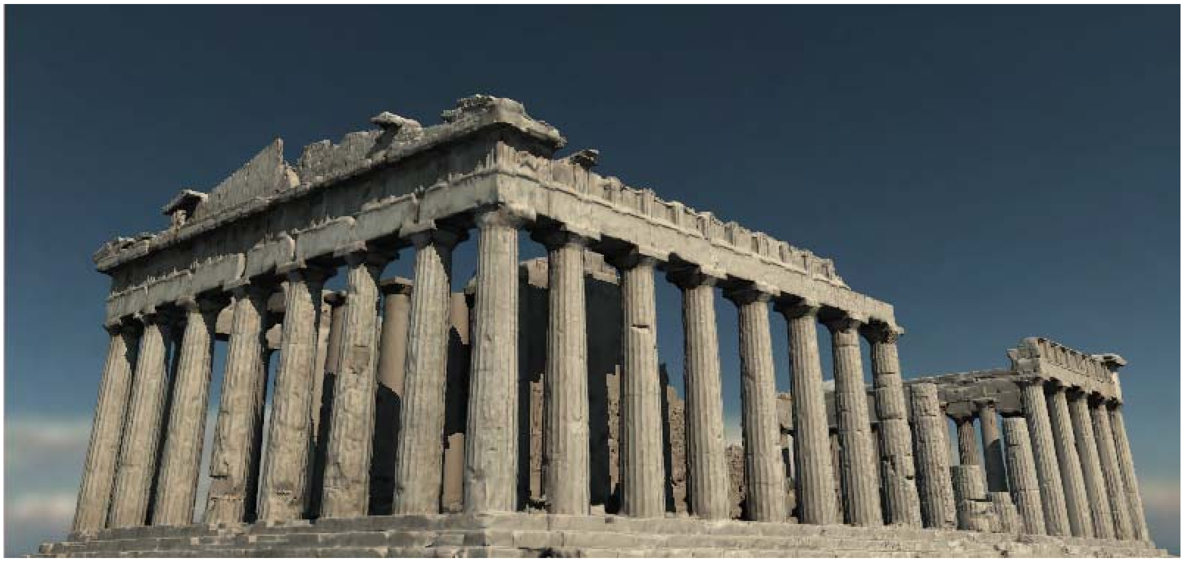


Figure 1. *A real-time rendering of the Parthenon.*

2.1 Overview

In this chapter we briefly describe techniques used represent and render the high dynamic range (HDR) time-lapse sky imagery in the real-time Parthenon demo (Figure 1). These methods, along with several other rendering techniques, achieve real-time frame-rates using the latest generation of graphics hardware.

[3] jisidoro@ati.com
[4] psander@ati.com

2.2 The algorithm

Efficient storage and rendering of a high-dynamic range animated skybox is a significant challenge for real-time applications. In our application, the sky is represented by a series of 696 panoramic time-lapse sky images, consisting of 2.17 GB of compressed HDR imagery (Figure 2). Since the target is to share the 256 MB of video memory with the rest of the demo, we need to develop a method to compress this data.

Preprocess. The first step of our approach is to decompose the sky video into a sequence of reduced range sky images with low-frequency HDR information. This is accomplished by computing 3rd order spherical harmonic coefficients for each image, resulting in a low-frequency representation (Figure 3), and then dividing the HDR image by the low-frequency result and storing it using 8 bits per channel. The reduced range imagery can then leverage standard video codecs such as XVid or DivX.

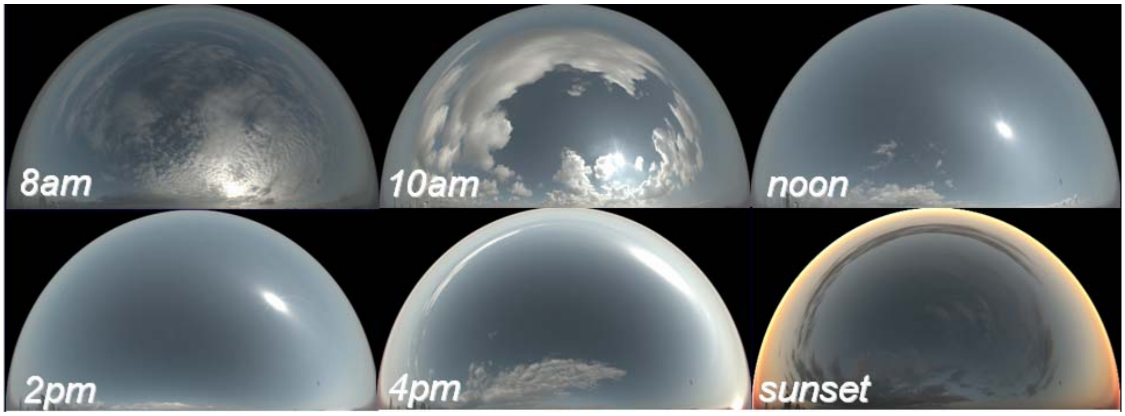


Figure 2. *Six frames from the animated sky sequence.*

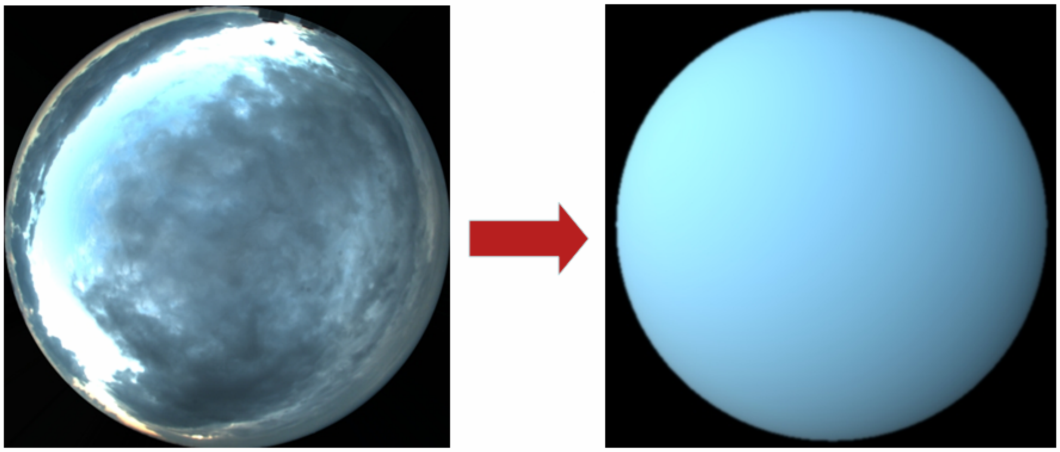


Figure 3. *From the original HDR sky image, a low-frequency approximation based on spherical harmonics is computed.*

Runtime. In order to render the skybox, a reduced range video sequence is played back into a texture that is mapped to the hemisphere.

The low-frequency HDR spherical harmonics coefficients are used in three ways. The first usage is to re-multiply the reduced range imagery by the HDR data at runtime in order to regenerate the high dynamic range imagery.

The second usage of the low-frequency HDR spherical harmonics data is to provide ambient illumination for any given time of the day, as illustrated in Figure 4. This is achieved by looking up the color of the light coming from the sky in the direction of the surface’s bent normal. For direct illumination, we precompute and use a per-frame sun color that accounts for occlusion by clouds and reddening of the sun in the evening sky. The resulting illumination at different times of the day is shown in Figure 5.

The third usage is to provide a smooth transition between the distant terrain and the sky. We achieve a realistic horizon rendering result by blending between the reconstructed sky data and the approximation using spherical harmonics as we approach the horizon. Below the horizon, only the spherical harmonics approximation is used. This allows us to render the far terrain using alpha-blending near the horizon and get a smooth transition between land and sky as shown in Figure 6.

Optic flow. The running time for the demo is roughly 120 seconds, so for 690 frames of data, the video frame-rate would only be about six frames per second if it were expanded to the full length of the video sequence. To make the playback smoother, we pre-compute the inter-frame optic flow, in order to morph each frame of video into the next using the skybox pixel shader at runtime. Since the cloud motion field is relatively low-frequency for most of the video sequence, the optic flow information is stored as 16x16 images in slices of a 3D texture. Using the 3D texture allows for smooth interpolation of the flow-fields between subsequent frames.

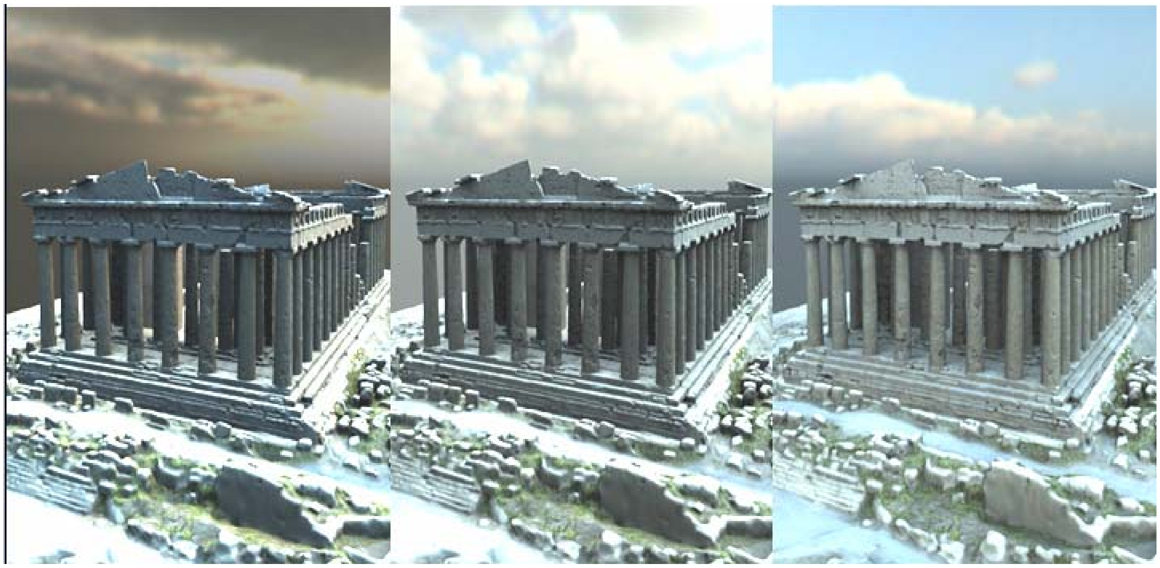


Figure 4 *The ambient lighting at different times of the day.*

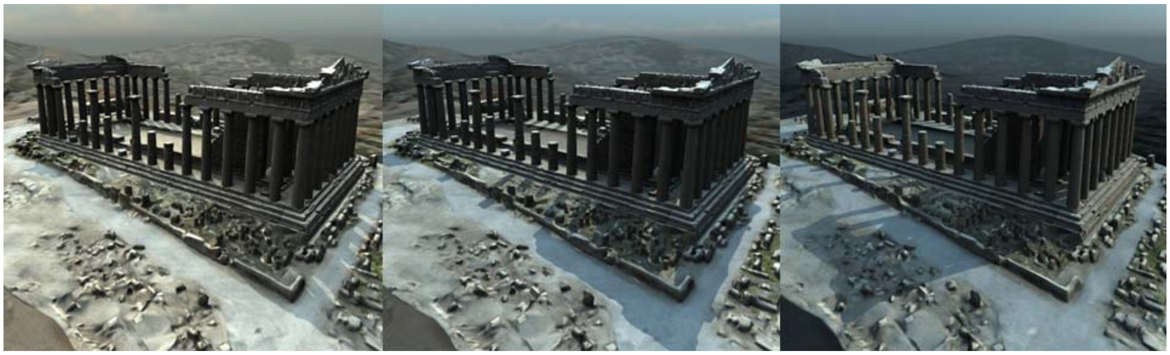


Figure 5 *The final rendered result, including direct illumination and shadow mapping at different times of the day.*

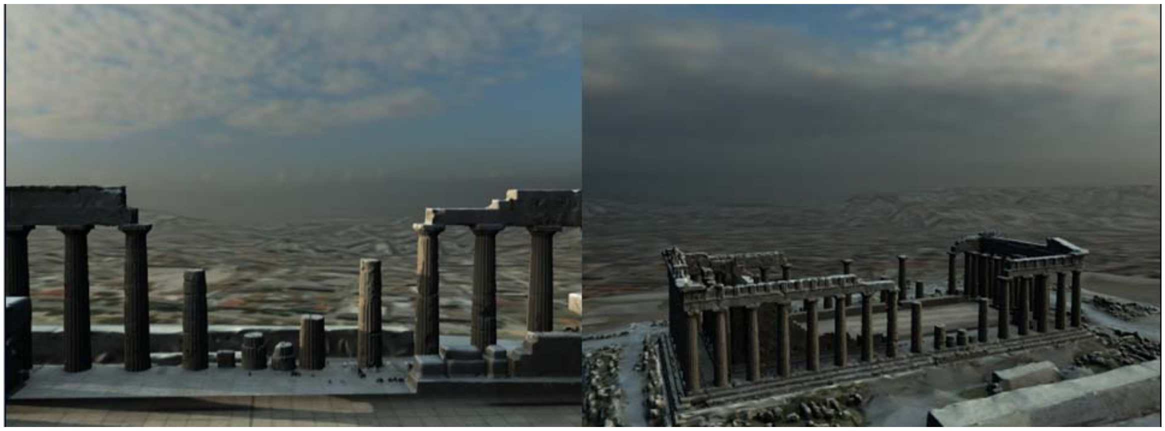


Figure 6*. Approximate horizon rendering by alpha-blending the terrain over the spherical harmonics approximation of the sky near the horizon.*

2.3 Summary

To summarize, we presented methods for efficiently storing and rendering an animated skybox. The method decomposes the sky imagery and stores it using an 8-bit high-frequency video along with 3rd order spherical harmonics coefficients to capture low-frequency HDR data. We also show how we use this spherical harmonics approximation in order to light the scene.

2.4 Acknowledgements

We would like to thank Paul Debevec and his team at USC for the original Parthenon dataset and sky imagery. We thank Eli Turner for his help with the artwork. Finally, we thank the members of ATI’s 3D Application Research Group for insightful discussions and suggestions.

Chapter 3

Artist-Directable Real-Time Rain Rendering in City Environments

Natalya Tatarchuk[5]
ATI Research

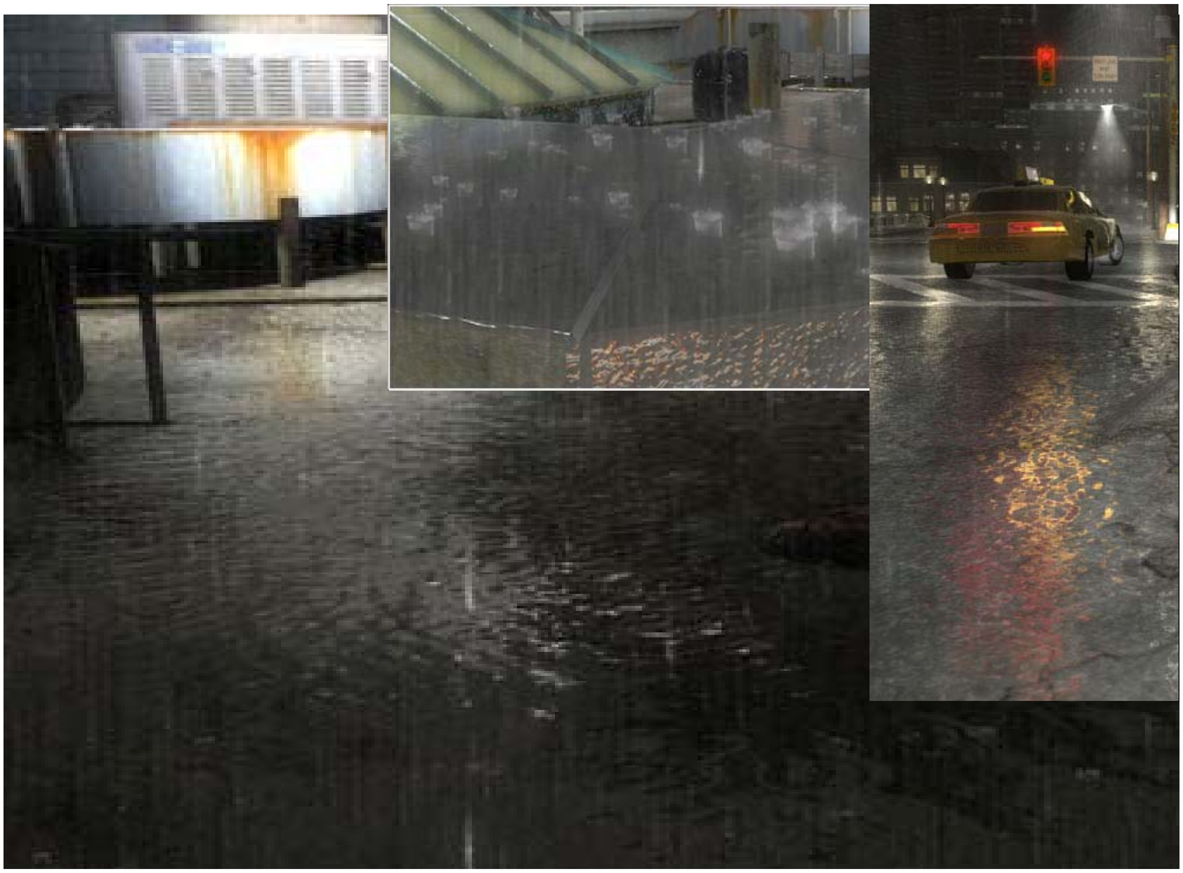


Figure 1. Various rain effects seen in the state-of-the-art interactive demo "ToyShop": dynamic water puddle rendering with raindrops, raindrop splashes, and wet view-dependent reflections of bright light sources.

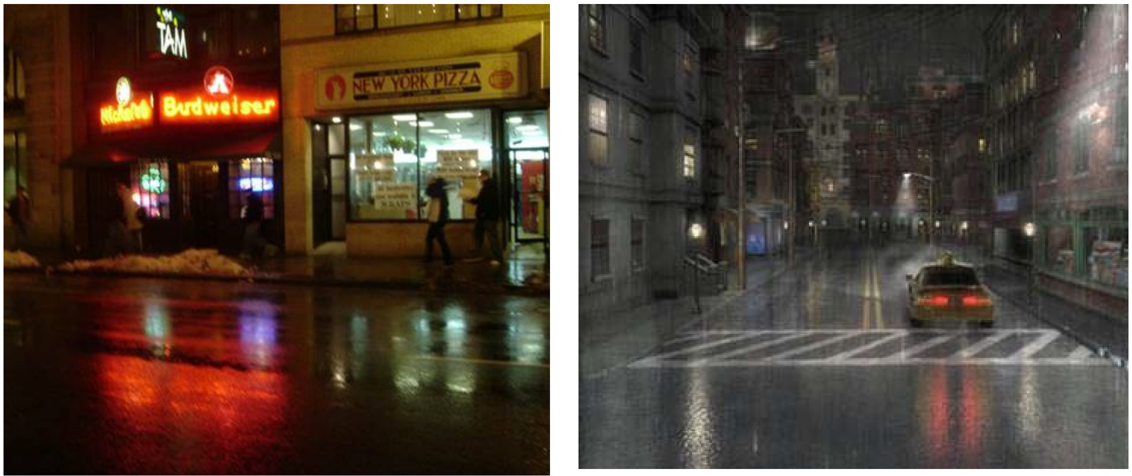
[5] natasha@ati.com

3.1 Abstract

In this chapter we will cover approaches for creating visually complex, rich interactive environments as a case study of developing the world of ATI “ToyShop” demo. We will discuss the constraints for developing large immersive worlds in real-time, and go over the considerations for developing lighting environments for such scene rendering. Rain-specific effects in city environments will be presented. We will overview the lightning system used to create illumination from the lightning flashes, the high dynamic range rendering techniques used, various approaches for rendering rain effects and dynamic water simulation on the GPU. Methods for rendering reflections in real-time will be illustrated. Additionally, a number of specific material shaders for enhancing the feel of the rainy urban environment will be examined.

3.2 Introduction

Our goal was to create a moment in a dark city, downtown, during a rainy night. Fortunately, we had many opportunities for research, having started on the concept for the demo in the middle of October in Boston. As a comparison, in Figure 2 we see a snapshot of the theater district in Boston downtown compared with the final rendering in the ToyShop demo.



(a) Rainy night in downtown Boston *(b)Rainy night it the ToyShop town*

Figure 2. *Comparison of a photograph from a real city during a rainy night versus a synthetic rendering of the interactive environment of ToyShop.*

Some games which incorporate rain rendering in their worlds use a very straight-forward approach: rendering stretched, alpha-blended particles to simulate falling raindrops. This approach fails to create a truly convincing and interesting rain impression. Frequently, the games only include one or two effects such as the stretched rain particles and perhaps a simple CPU-based water puddle animation to simulate the impression of rainy environment. This results in an unrealistic rendering with the rain not reacting accurately to scene illumination, such as lightning or spotlights. Some notable exceptions to this are the recently released *Spinter Cell* from Ubisoft and *Need for Speed: Most Wanted* (Xbox 360) with much improved rain rendering.

We present a number of novel techniques for rendering both particle-based rain and rain based on a post-processing image-space technique, as well as many additional secondary rain effects, without which one cannot generate an immersive rain environment. Rain is a very complex atmospheric physical phenomenon and consists of numerous visual cues:

- Strong rainfall
- Falling raindrops dripping off objects' surfaces
- Raindrop splashes and splatters
- Various reflections in surface materials and puddles
- Misty halos around bright lights and objects due to light scattering and rain precipitation
- Water, streaming off objects and on the streets
- Atmospheric light attenuation
- Water ripples and puddles on the streets

We have developed methods to render all of the above in a real-time environment. Our techniques provide a variety of artist-directable controls and respect the rules of physics for simulating rainfall. They utilize light reflection models to allow the rain to respond dynamically and correctly to the lighting changes in the complex environment of the ATI "ToyShop" demo due to illumination from atmospheric effects (such as lightning).

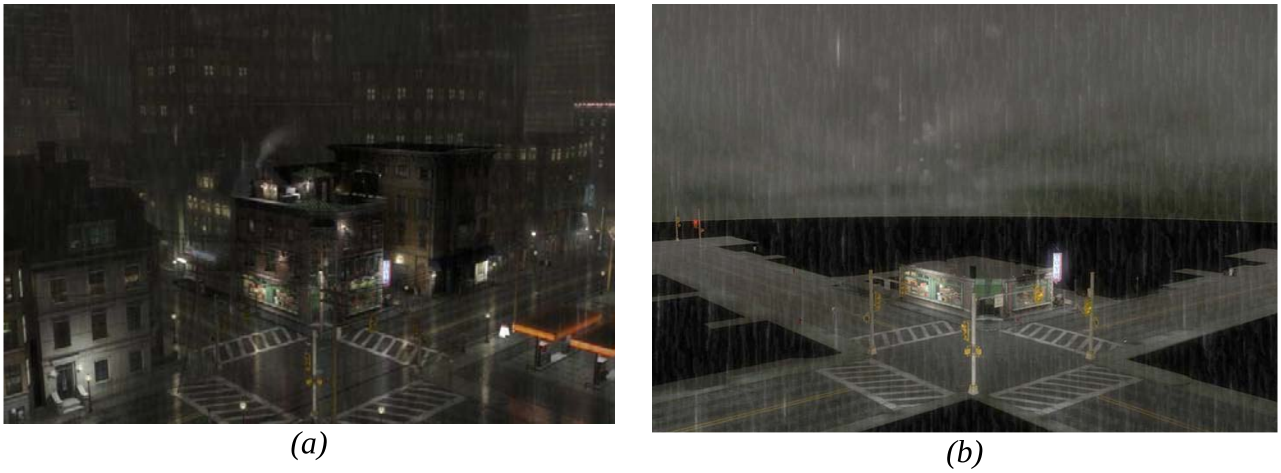
3.3 Rendering system requirements and constraints

We faced a number of strict constraints while developing this interactive environment. First and foremost, the memory consumption needed to be in check – the goal was to fit the entire world of this environment into less than 256 MB of video memory. This was required for optimal performance on any graphics hardware, although this demo was specifically targeted to push the limitations of ATI Radeon X1800 XT graphics card. This memory requirement was a severe constraint for our development due to the large scope of the environment we wanted to create. At the end of the production we managed to fit the entire assets into 240 MB including 54 MB memory used for back buffer and offscreen storage, 156 MB for texture memory (including many high resolution textures) and 28 MB for vertex and index buffers.

We must note that the high performance interactivity of this environment would not be achievable without using 3Dc texture compression. In order to create a realistic environment, we used a great deal of high resolution textures to capture the extreme detail of the represented world (for example, light maps and lightning maps, high resolution normal maps and height maps, high resolution color maps and specular maps). Using 3Dc technology allowed us to compress nearly half of a gigabyte of texture assets (478 MB, to be exact) to 156 MB. Specifically, we used texture compression formats such as ATI2N and ATI1N, DXT1 and DXT5 to compress majority of our textures. Additionally, we recommend using vertex data format DEC3N, which allows 10-bit per-channel data storage, to reduce memory footprint for geometry. This format gives

3:1 memory savings while maintaining reasonable precision. We used it for encoding vertex normal, tangent and binormal data, as well as some miscellaneous vertex data. Note that this format is available on a wide range of consumer hardware and will be included as a first-class citizen in the upcoming DirectX10 API.

We would like to provide a visual comparison of what the usage of 3Dc texture compression technology and the vertex compression format DEC3N allowed us to do in our interactive environment: in Figure 3a, the scene is rendered using compressed formats and in Figure 3b, only a portion of the original scene is rendered. This is due to the fact that the entire scene would simply not fit into the graphics card memory, and thus would not be rendered in real-time. Although this example is a bit contrived (since in real-life productions, one would first reduce the resolution of texture maps and decimate the geometric meshes), it serves the purpose of stressing the importance of high quality compression technology for interactive environments.



(a) *(b)*

Figure 3. *In (a) the entire scene is rendered with the use of 3Dc and DEC3N technology. Without using this technology, we are only able to fit a small portion of the original environment into the graphics card memory (b).*

With the advances in programmability of the latest graphics cards and recent innovative games such as FarCry® by Crytek and Half-Life 2 by Valve Software, everyone realizes the need and the desire to create immersive interactive environments. In such worlds a player has many options to explore the surroundings; and the complexity of the environments helps make the experience truly multifaceted. However, that said, rich detailed worlds require complex shaders, and require a variety of those.

In order to create the imagery of the ToyShop environment, we have developed a number of custom unique material and effects shaders (more than 500 unique shaders). Of course, approximately half were dedicated to rendering rain-related effects, including dynamic water simulations and wet surface material shaders. Just about one third of the entire shader database was used for rendering depth information or rendering proxy geometry objects into reflection buffers (see section 3.5). Finally, a number of shaders (roughly one sixth) were dedicated to rendering post-processing effects such as glow and blurring, along with the many custom particle-based effects. We highly recommend developing an extensive include framework for your projects. Separating shader sections responsible for rendering specific materials or effects allowed us faster iteration. For example, we used more than 20 include files containing functions to compute reflections, shadows using shadow mapping, high dynamic range rendering including HDR lightmap decoding and tone-mapping, mathematical helper functions and lighting helper functions. In essence the include file system allows a game developer to emulate the richness of shader programming languages such as RenderMan®.

3.3 Lighting system and high dynamic range rendering

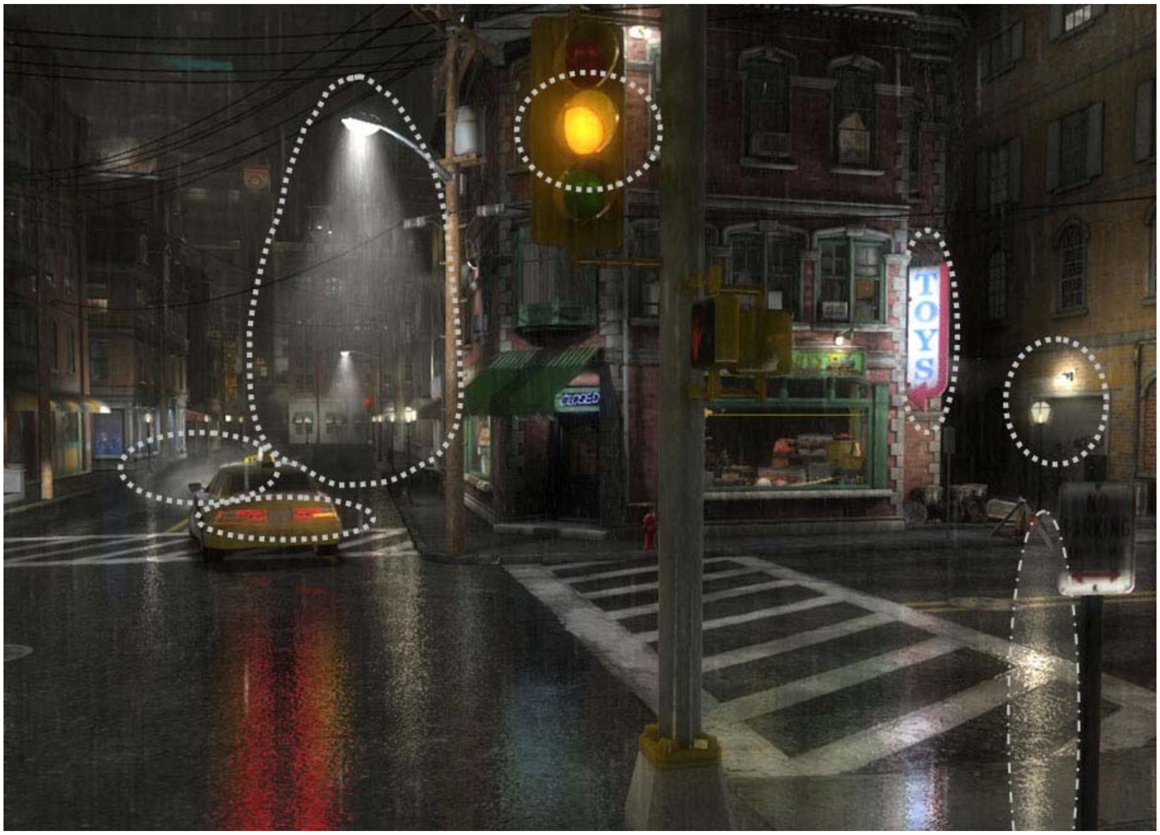


Figure 4. *In the dark stormy night in the ToyShop town, we see a large number of apparent light sources*

We set out to create a moment in a dark city in a stormy, rainy night. But even in the midst of a night there is light in our somber downtown corner. Notice the large number of perceived light sources visible while flying around the environment (see Figure 4), such as the bright street lights, the blinking neon sign on the corner of the toy shop, various street lamps and car head and tail lights, and, last but not least, the lightning flashes. Every bright light source displays reflections in various wet materials found in this city. The sky also appears to illuminate various objects, including the raindrops and their splashes in the street puddles. While there are a number of different approaches that were used for implementing specific effects, an overall lighting system was developed in order to create a consistent look.

3.3.1 HDR rendering on a budget

Any current state-of-the-art rendering must be done using high dynamic range for lighting and rendering[6]. Without using this range of colors, the resulting lighting in the scene will feel dull. (See chapter 7, section 7.5 for an excellent review of the HDR system in Valve's Source engine). However, considering the memory constraints placed on this production, strict care had to be taken with regards to the specific format and precision selection for the rendering buffers. Additionally, since every rendered pixel on the screen goes through the tone-mapping process, the choice of the tone-mapping function directly affects the performance. Depending on the specific selection, it can constitute a significant performance hit. Because of the multitude of effects desired for the immersive environment rendering, both memory requirements and performance considerations were very specific and stringent.

The goal lied in balancing performance and memory usage with an expanded dynamic range and good precision results. Recent graphics hardware such as ATI Radeon X800 and above provides access to renderable surfaces which have 10 bits per channel. This surface format provides excellent precision results at half the memory usage of 16 bit floating point formats. All of the back buffers and auxiliary buffers were created with this surface format, as well as the HDR lightmaps used to light the environment.

One challenge with using HDR for rendering in production environments currently lies in the scarcity of publicly available tools to work with the HDR art assets. Aside from previewing the lightmaps in-engine in real-time, there was a considerable difficulty for visualizing the lightmaps outside of the engine. We recommend using [HDRShop] for visualization of the light maps early on, as they take a considerable amount of time to render and ideally should not be rendered multiple times.

3.3.2 Tone mapping and authoring HDR lightmaps

For our HDR lightmap decoding, we used the RGBS (fixed-point RGB with shader Scale) approach for HDR lightmap decoding in shaders to maximize the available dynamic range of the 10 bit surface format. Simply using the straight 10 bit format for encoding and decoding light information in an expanded range results in the range expanded from [0, 1] to [0, 4]. However, if we are able to use the extra two alpha bits as a shared exponent, the range can be stretched to [0, 16]. See [Persson06] for a very thorough overview of available texture formats in the context of HDR rendering.

The tone mapping curve was expressed as a four-point artist-editable spline for more control of the lighting. Some suggestions for integrating HDR into a game engine or a production pipeline:

- Start by applying a linear tone mapping curve

[6] For an excellent introduction and overview of image-based lighting and high dynamic range rendering, see [Reinhard05].

- This allows you to make sure that the values that you are getting from lightmaps in engine are correlated to the actual stored values in lightmaps
- Remember – lightmaps are extremely time-consuming to render (especially if using any precomputed global illumination effects). Ideally the pipeline should be thoroughly tested prior to starting the process of lightmap rendering. That means testing the tone mapping curves first for accuracy

Let's look at some examples of HDR lightmaps in the context of HDR rendering and tone mapping. Figure 5 shows an example of the tone mapping curve incorrectly darkening and significantly reducing the available range for lighting. Therefore all details in the shadows are completely lost.

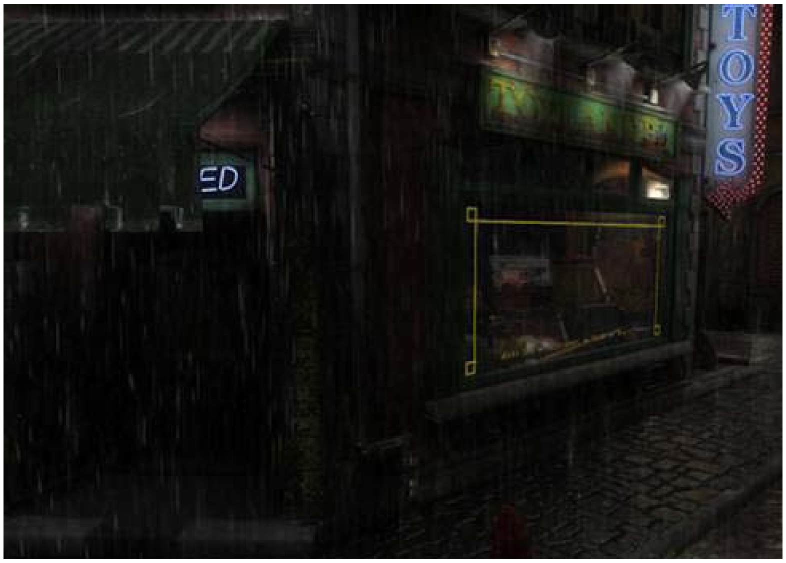


Figure 5*. Example of a tone mapping curve set to excessively darken the scene*

In Figure 6 we are seeing the opposite effect, where the tone mapping curve is over - saturating the scene and all of the details in the bright regions are blown out and lost.

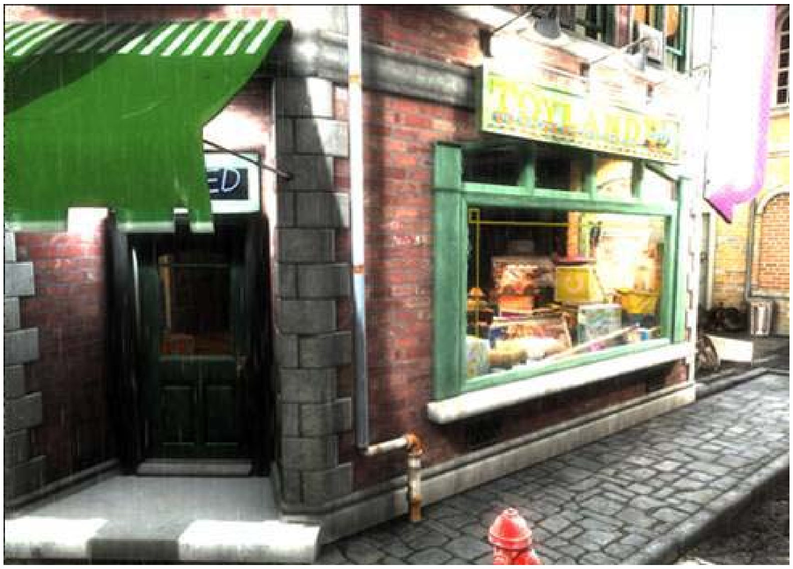


Figure 6*. Example of a tone mapping curve set to excessively oversaturate the scene*

We can see the tone mapping used in our interactive environment in Figure 7 where the contrast is set to the artistic choice. In essence, the tone mapping process comes down to being an aesthetic preference, and all of the three figures will look appealing in some circumstances and undesirable in many others. Additionally, the tone mapping curves can be dynamically animated depending on the scene intensity, location and timing. For an example of dynamic tone mapping, see chapter 7 of this course for the description of the HDR system in Valve's Source engine.

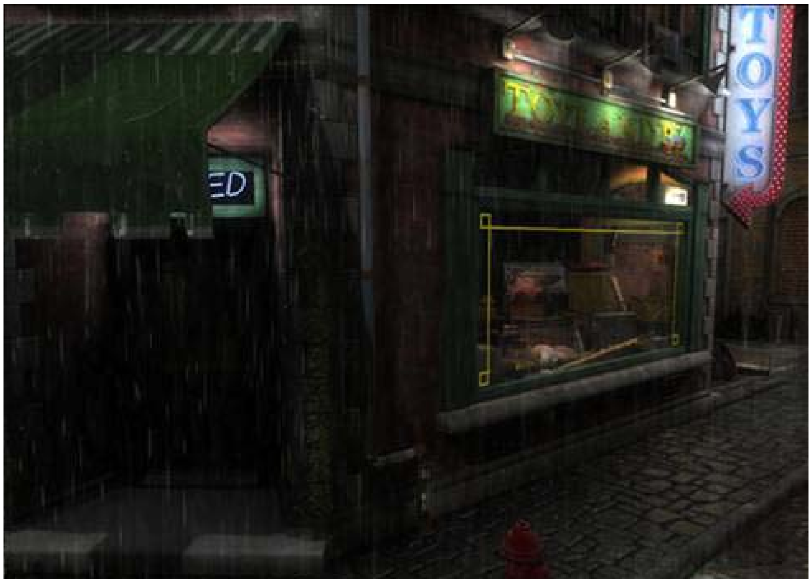


Figure 7*. The tone mapping settings used in our environment allowed us to preserve desired amount of details in shadowed areas while maintaining overall contrast in the scene. Note the details under the "Closed" sign of the store.*

With respect to authoring lightmaps for any interactive rendering, the key notion to remember is that the process of rendering lightmaps traditionally is rather time consuming (in terms of the rendering time), since it is an offline process. Thus the goal is to postpone this to the later parts of the development after iterating on some test lightmaps as a proof of concept. And, naturally, rendering lightmaps can only happen when the scene geometry and texturing, along with the lighting setup, has been completed and approved for the final environment.

We chose to approximate global illumination effects with a less computationally-intensive approach. Due to time constraints and hardware limitations, we could not render both the final gather and global illumination in Maya® 6.0 Mental Ray for the lightmaps. However, the visual results of global illumination are crucial for the impression of light bouncing off nearby objects, and we needed a method to approximate it in the lightmaps. As an alternative, we chose to render lightmaps into a 32 bit texture format with final gather only using HDR light sources in Maya® (however, the light intensities for these light sources were set such that the final lighting values never exceed 16). See section 3.3.1 for the description of our HDR lighting and rendering setup. Additionally, the light intensities varied depending on the falloff type and distance from surfaces. An example of such a resulting lightmap render is in Figure 8a.

To help approximate the soft lighting of global illumination we rendered the ambient occlusion map (in Figure 8b) into a 32 bit grayscale texture. This texture can be thought of as a global illumination noise map, to provide us with the darkening of the lighting in objects' creases and in the areas where objects meet. To combine the ambient occlusion map and the prerendered lightmap, we used the Digital Fusion software package, first applying tone mapping to the ambient occlusion map to move the values into the appropriate range for the rendered lightmap and then adding it to the lightmap texture. The resulting final lightmap in Figure 8c displays an example of the actual art asset used in the interactive environment. Note that we are able to preserve both the hard contact shadows and the soft bounced lighting in the resulting lightmap.

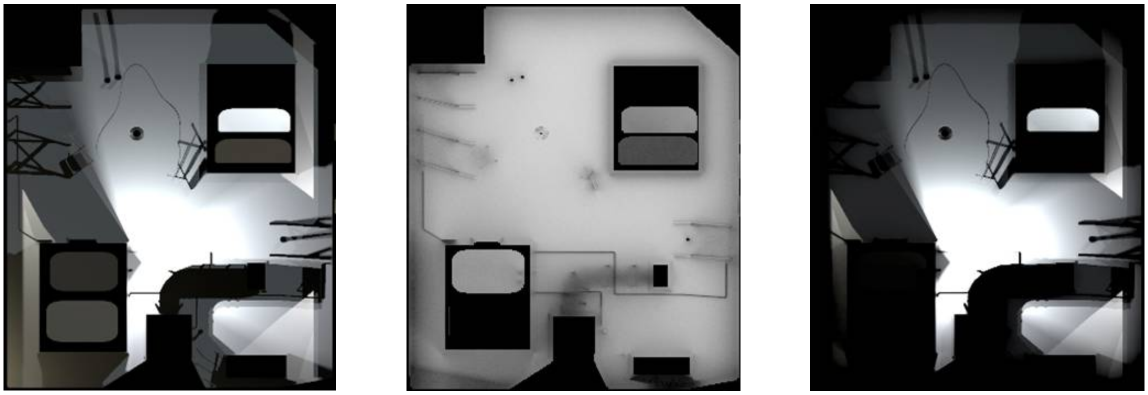

(a) (b) (c)

Figure 8. *An example of the lightmap asset creation process. We start out with the lightmap rendered with the final gather process in Maya® 6.0 Mental Ray (a), and then combine it with the ambient occlusion map from the same environment (b) to create the final art asset (c) used in the real-time environment.*

3.3.3 Shadowing system

In the heart of the night, shadows rule the darkness. Enhancing the dynamism of the immersive environment of our interactive city meant having support for varied light conditions including dynamic lights. The shadow mapping technology allowed us a good balance between the final performance and the resulting quality of dynamic soft shadows. Shadow mapping refers to the process of generating shadows by rendering from the point of view of the light source and then using the resulting texture (called the *shadow map*) during the shading phase to darken material properties depending on whether they are occluded with the respect to a specific light source. For an excellent overview of the shadow mapping and other shadowing techniques see chapter 6 in [Akenine-Möller02].

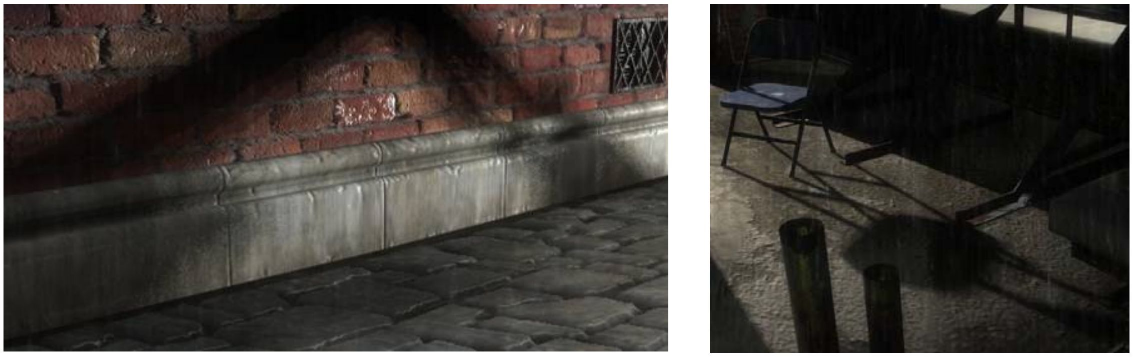

(a) *(b)*

Figure 9. *Examples of shadow mapping technology used to render dynamic shadows on varied surfaces in the ToyShop demo*

We used a single animated shadowing light for our environment, which resulted in a single shadow map. To conserve memory, we took advantage of the novel hardware feature in the ATI Radeon series called ‘depth textures’. These are efficient and flexible surfaces for shadow mapping and other algorithms (see [Isidoro06] for a meticulous overview of these hardware features, and the efficient methods to implement shadow mapping and soft shadow filtering). Since these surface formats do not mandate binding of an identical sized color buffer, they provide excellent savings in memory for depth-only rendering (we were able to bind a 1x1 color buffer to a 1024x1024 shadow map). A 16-bit DF16 texture format provided good precision for our depth complexity needs. In Figures 9a and 9b we see some examples of shadow mapping applied in our environment.

Shadow mapping has several important considerations that any developer needs to address in order to obtain visually pleasing results. The actual resolution of the shadow map will be directly proportional to the appearance of the shadows (as well as affect the performance). If the shadow map resolution is not high enough for the desired scene and viewer closeness to the shaded objects, potential aliasing artifacts may appear. We used a relatively large shadow map of 1024x1024 to improve the results. Additionally, a common bane of shadow maps is the need to filter the penumbral regions of the shadows in order to create soft shadows without aliasing artifacts. A typical approach includes using percentage closer filtering (PCF) ([Reeves87]) to soften the shadow edges. However, simple PCF does not typically yield visually satisfactory results without aliasing artifacts or visible filtering patterns. We improve on the approach by using a randomized PCF offset sampling with custom filtering kernels ([Isidoro06]). For bright materials (or regions, such as in Figure 9a) we used an 8-tap Poisson disk kernel with random rotation offsets encoded into a lookup texture. Random rotation of the sampling kernel is needed for bright surfaces with smooth albedo, since the shadowing artifacts are most visible for these materials. For dark and noisy textured surfaces, we can relax the sampling requirement and can use a fixed Poisson disk kernel for sampling the shadow maps (as in Figure 9b).

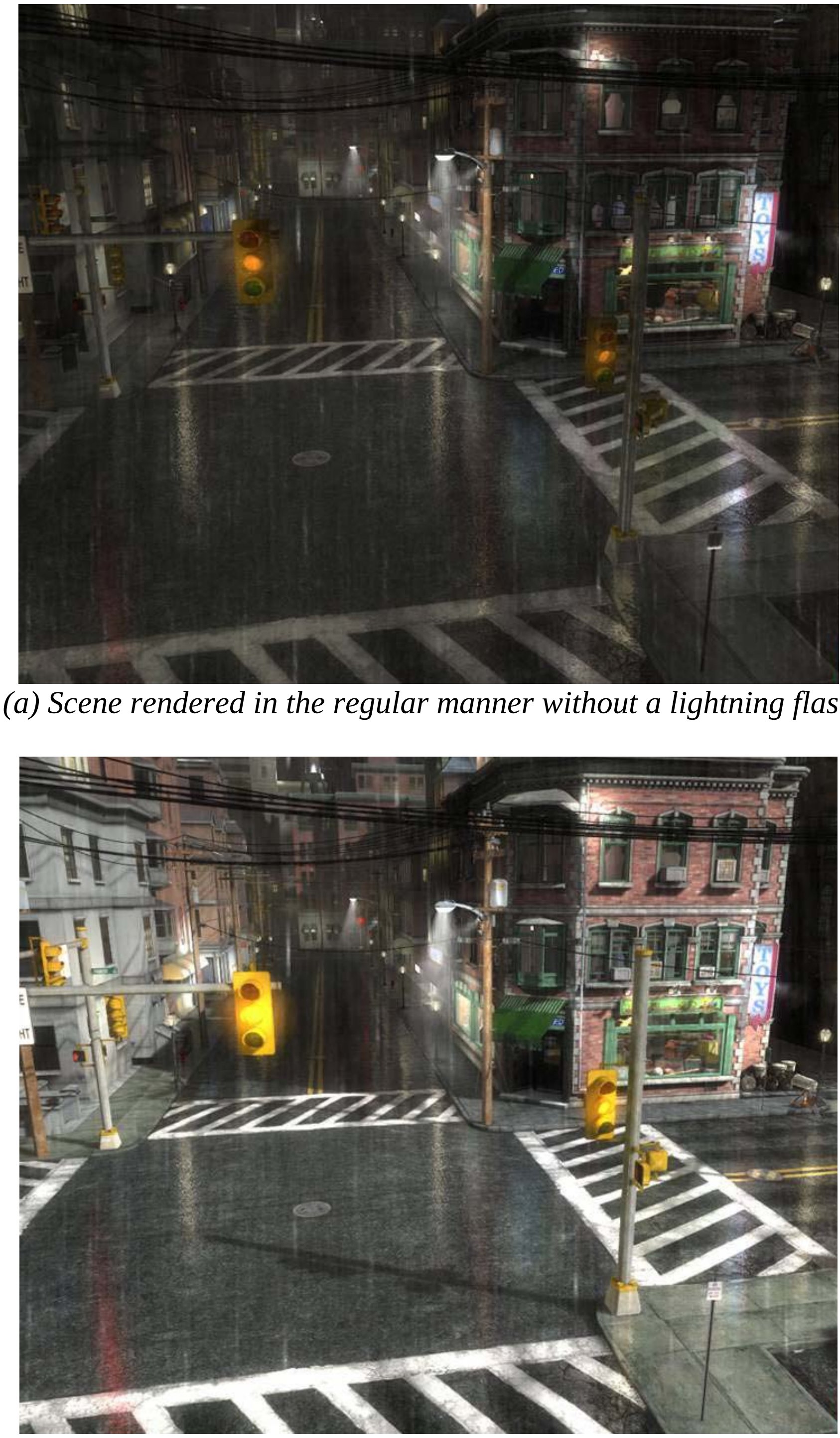
3.3.4 Lightning System and Integration

Once we have the lighting and shadowing systems in place, we can start putting together the other components in order to create the illusion of a dark stormy night. Lightning and thunder increase the feel of a rainy, turbulent environment. Illumination from the lightning flashes needs to affect every object in the scene. As such, uniformly aligned shadows are crucial – we need to create a feeling that somewhere in the sky, a lighting bolt just struck in a particular location. At the same time, we have to keep in mind that we are using a single shadowing light and a single shadow map for our shadowing solution. Computing lightning shadows for each additional lightning 'light' can significantly impact performance and memory footprint (with additional shadow maps).

In Figure 10a the city corner is rendered with just the regular lighting and shadowing system. In Figure 10b we captured the effect of a lightning flash on our environment – note all of the objects are throwing uniformly aligned shadows onto the street, creating a strong impression of actual lightning flash (combined with the sound effects of thunder).

Lightning is a strong directional light that affects every object in the scene. Creating a convincing and realistic lightning effect is challenging for a variety of reasons. In our interactive environment, the viewer can get very close to the objects when lightning strikes. That means that the resolution of the shadows generated by the lightning flash must hold up regardless of viewer's distance from an object. Finally, the illumination from the lightning must seamlessly integrate into the main lighting solution for our environment, as we are using a consistent illumination model for all objects in our scene (including shadow mapping).

For artistic reasons, we felt it was important to include lightning illumination in our environment, despite the challenges – it heightens the mood. A dark night with rough weather would not affect the viewer in the same manner without the sudden surprise of a lightning flash followed by the inevitable thunder. Additionally, the extra illumination from the lightning helped us show off the details of various effects in the scene, which may have gone unnoticed in the pure darkness of the night.



(a) Scene rendered in the regular manner without a lightning flash

(b) Scene rendered during a lightning flash

Figure 10. *Comparison of the regular environment rendering in (a) versus the same environment lit by a lightning strike from the right (b). Notice the correctly aligned object shadows in (b).*

Our solution lies in special *lightning lightmaps* for the illumination due to lightning flashes. We can prerender the result of illuminating the environment from several directions, mimicking the light from a lightning flash into a lightning lightmap texture. Unlike a regular lightmap, this texture does not need to store full lighting color information – we are only planning to use it to modulate the regular illumination computed for each pixel (as an intensity multiplier of the underlying HDR lightmap). Therefore we simply encode the value into a single channel 8 bit texture. In our case, we found that computing the illumination for two unique lightning light locations was sufficient and provided good results for the additional increase in memory consumption (as a two-channel 8-bit-per-channel lightning lightmap, example in Figure 11). The scene information is encoded in a manner similar to regular lightmaps. We provide the artists an editable intensity parameter for custom mixing of the two lightmaps – which can be animated and controlled dynamically on a per object basis by a rendering script in our engine (we use the Lua programming language). The first lightmap contained the illumination from a lightning flash at an angle from far away, and the second lightning lightmap contained the illumination from a lightning flash directly above the center of the scene. Mixing these two maps in different object- and time- specific ways creates an illusion that we have a wider variety of lightning flash directions that we actually did.

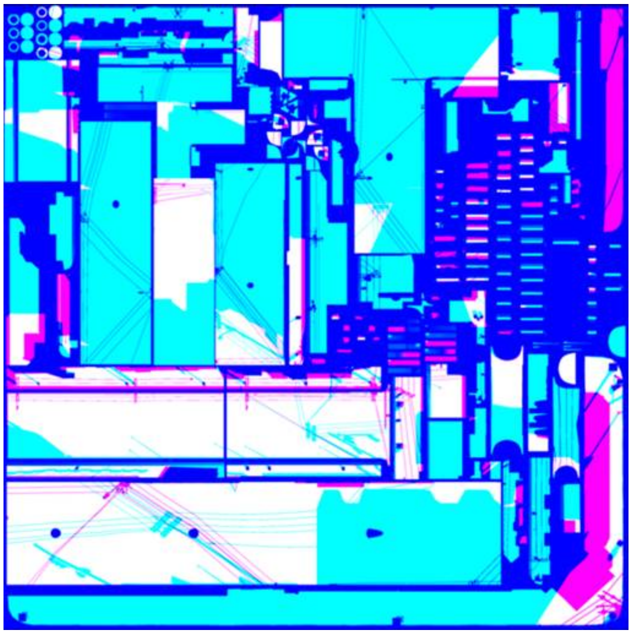


Figure 11. *An example of a lightning lightmap where an individual lightning intensity value is stored for two lightning light locations in red and blue channels of the texture.*

Every shaded pixel in our environment uses lightning illumination information. The rendering script propagates the animation parameter for each of the two lightning flashes to all of the shaders in the form of uniform parameters (floating point value of lightning brightness and location). In a specific material shader we can either read the lightning lightmap for the intensity value for the specific lightning selection or simply use the lightning brightness parameter (controlled by the artists from outside the script). (Or both types of parameters can be used simultaneously). The lightning lightmap sample is added to the regular lightmap sample before tone mapping. The performance cost for integrating this type of lightning illumination computation is very low – a single texture fetch plus several ALU operations in the shader to compute lightning flashes from varied locations. All objects in our real-time environment use this scheme and thus appear to respond accurately to lightning illumination in the scene.

Additionally, note that in realistic scenes, translucency of water is affected by the lightning flash illumination. We mimic this effect in our rendering. This can be accomplished by using the lightning brightness value to adjust the pixel's opacity (alpha value) when the lightning flash occurs. We use this approach extensively in the rain effects, improving the visual quality of those effects and making the lightning flash appear more natural.

3.4 Post-processing effects for rain rendering

In recent years post-processing has become a popular approach for adding visual variety to games, as well as approximate many camera or environment properties of the world around us. For example, the post-processing pipeline is used to add the depth-of-field effects (as described in [Riguer03] and [Scheuermann04]), enable high dynamic range rendering by providing a tone mapping step in the end of the scene processing, various image processing for artistic effects (some examples of post processing in a game environment are covered in chapter 7 of this course). In the Toyshop demo we used the flexible post-processing pipeline available in our engine to approximate atmospheric effects such as misty glow due to light scattering, to perform tone mapping for HDR rendering and for a variety of specific blurring effects for creation of rain effects.

3.3.1 Creating appearance of misty glow due to inclement weather

Water particles in the atmosphere during the rain increase the amount of light scattering around objects. Multiple scattering effects are responsible for the appearance of glow around light sources in stormy weather ([Van de Hulst81]). In order to approximate the effect of halos around bright light sources, we make use of the post-processing pipeline available in our engine and controllable through the rendering script. See Figure 12 below for an example of misty glow in our environment.

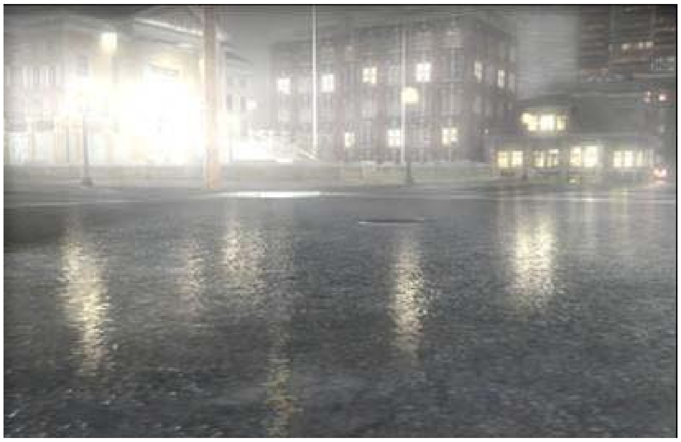


Figure 12. *Misty glow in the ToyShop environment*

To approximate the atmospheric point spread function which can be used to model multiple scattering around the light sources in the stormy weather (as in [Narasimhan03]), we use the Kawase post-processing approach for rendering glows in our scene ([Kawase03]). The main concept lies in blurring the original image to create the glow halos around the objects and bright light sources. Blur is a ‘magic’ tool: it adds softness to the scene, and successfully hides some artifacts (similar to the depth of effects).

First we render our environment into an offscreen buffer, where the alpha channel is used to specify the amount of glow for each pixel. Since we use 10-10-10-2 buffers for rendering, we only use 2 bits of alpha for glow amount. This is not ideal for many scenes; however, with sufficient attention to details for material rendering we are able to achieve very good quality of the resulting effects even with just mere two bits of information and clever usage of blending states. Once the scene is rendered into an offscreen buffer (using 10-10-10-2 format), we downsample the rendering by a quarter in each dimension (giving a total of ¼ x ¼ = $^1/_{16}$ reduction). We apply small blur filters (shown in Figure 13 below) repeatedly to the downsampled image, performing four feedback ping-pong passes for computing blurring.

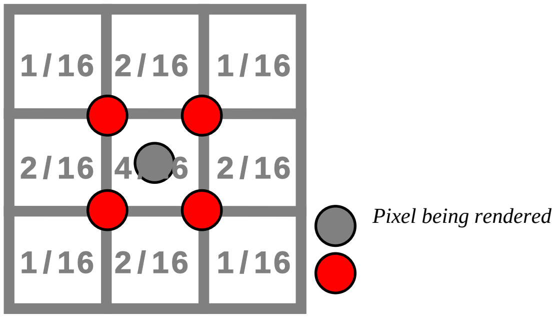


***Figure 13**. Kawase bloom filter. The weights for each sample are provided. (from [Kawase03]*

Each iteration of blurring 'ping-pongs' between two renderable textures used for storing the intermediate results. Each successive application of the bloom filter to the downsampled image takes the previous results as input and applies a new, larger kernel (as illustrated in Figure 14) to increase blurriness. The final blurring result is combined as described in [Kawase03]. More iterations will allow higher levels of blurriness; but we determined empirically that four passes give good visual results.

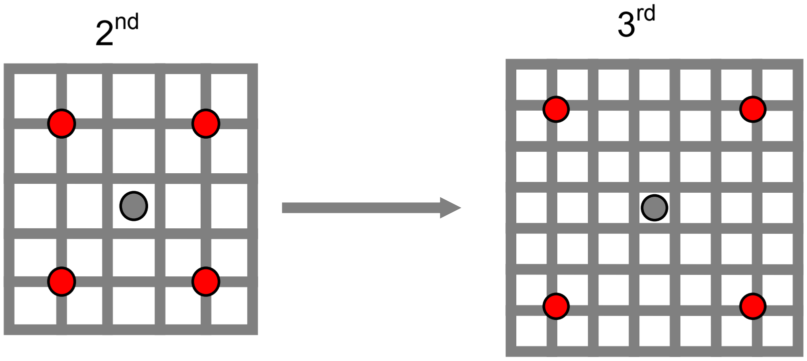


***Figure 14**. Two successive applications of the bloom filter on a texture grid. (from [Kawase03]*

To model fog attenuation due to water scattered in the atmosphere we implemented light attenuation based on distance in shaders. We attenuate the light information based on distance in shaders. In the vertex shader (Listing 1) we compute the distance of the object to the observer and then compute the linear fog value which is then sent to the interpolator for rasterization.

```
float4x4 mViewFog;
float2   vFogParams;

float ComputeFogFactor(float4 vWorldPos)
{
   // Compute distance to eye
   float4 vViewPos = mul (vWorldPos, mViewFog);
   float fDepth = sqrt(dot(vViewPos.xyz, vViewPos.xyz));

   // Compute linear fog = (d - end) / (end - start)
   float fFog = (fDepth - vFogParams.x) /
                (vFogParams.y - vFogParams.x);
   fFog = saturate(fFog);

   return fFog;
}
```

Listing 1. *Vertex shader fog segment*

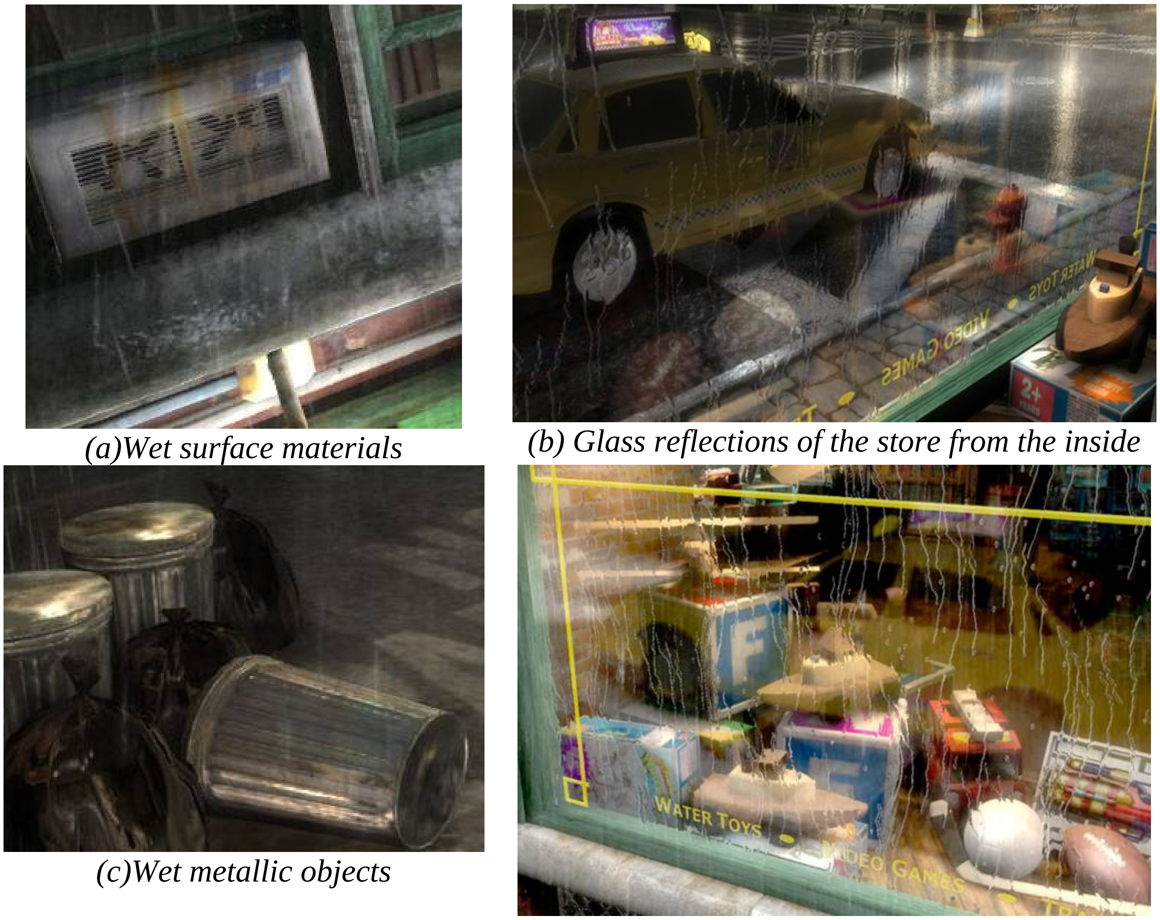
In the pixel shader (Listing 2), we use the computed and interpolated fog value to attenuate pixel color value before tone mapping.

```
float3 cFogColor;
float4 vFogParams;

float4 ComputeFoggedColor(float3 cFinalColor, // Pixel color
                          float glow, // Glow amount
                          float fFog) // Vertex shader computed fog
{
   float4 cOut;

   // Foggy factor
   float fogFactor = fFog * (1-(SiGetLuminance(cFinalColor)/10));
   fogFactor = min (fogFactor, vFogParams.z);

   // First figure out color
   cOut.rgb = lerp(cFinalColor, cFogColor, fogFactor);

   // Then alpha (which is the glow)
   cOut.a = lerp(glow, fogFactor*vFogParams.w + glow, fogFactor);

   return cOut;
}
```

Listing 2. *Pixel shader fog segment*

3.5 Wet reflective world

Realistic streaky reflections increase the feel of rain on wet streets and various object surfaces. These reflections are very prominent in any rainy scene and appear to stretch toward the viewer. Wet environments display a great deal of reflectivity – without realistic reflections the illusion is broken. Therefore, adding convincing reflections is a must for any rainy environment. To simulate the appearance of a wet city street in the rainy night, we render a number of various reflection effects in our scene:



(a)Wet surface materials

(b) Glass reflections of the store from the inside

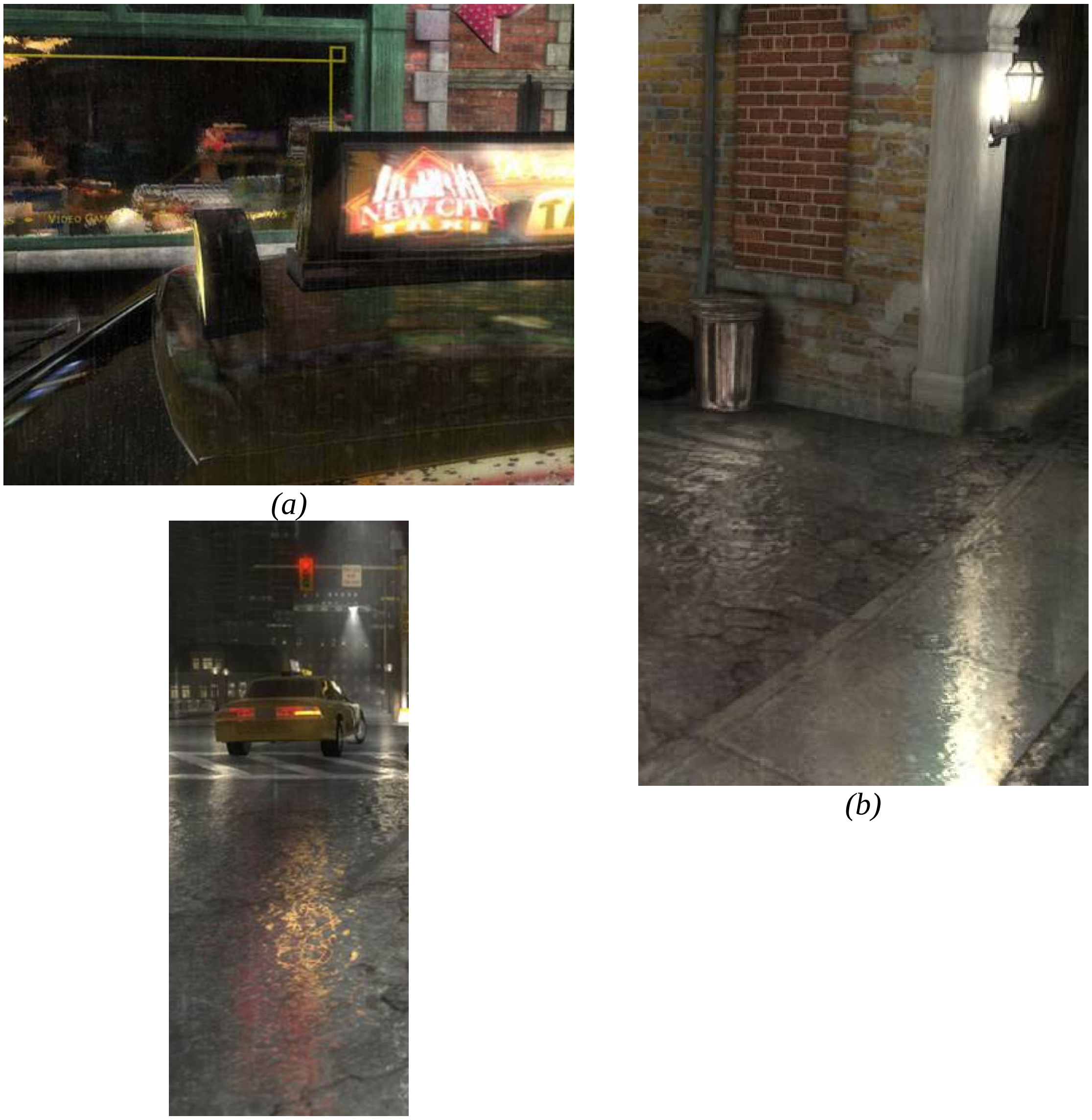
(c)Wet metallic objects

(d) Glass reflections from the outside as well as raindrop reflection

Figure 15. *A selection of reflection effects in the ToyShop environment.*

- Stretchy warped water reflections in the street, puddles and other wet surfaces (Figures 16b, 16c)
- Various wet surface materials (wet granite, pavement, cobblestones, plastic, metal grates, etc) (Figures 15a and 15c above)
- All of the rain effects used reflection and refraction effects (see section 3.6) (Figure 15d)
- The inside of the toy shore and the outside scene reflected in the glass panes of the store windows (Figures 15b and 15d)
- The drenched taxi cab turning around the corner displayed dynamic reflections of the scene around it (Figure 16a)

Depending on the polygonal properties of a particular object, highly specular surfaces can display a great deal of aliasing if one is not careful. We dedicated a significant amount of effort to ensuring that these artifacts are reduced, if not completely removed, from our interactive environment. The solution was to attenuate both reflection illumination and specular highlights at the objects' edges using a Fresnel term of varied powers.



(a)

(b)

(c)

Figure 16.*Dynamic reflection effects for rendering a wet taxi cab (a) and streets (b) in our interactive environment.*

3.5.1 View-Dependent Streaky Reflections

When moving around any city streets late night during a rain, one of rain's strongest visual cues are the stretched reflections of bright light sources (such as street lamps, cars lamps, and store signs in the streets and sidewalks). These reflections are very prominent in any rainy scene. They appear to stretch very strongly toward the viewer, distorting the original reflecting object vertically proportional to the distance from the viewer. Water in the puddles and on the streets further warp the reflections, increasing the feeling of wetness in the environment (especially during the actual rain, the falling raindrops hitting the puddles create dynamic warping of the reflections). It is also easy to notice that these types of reflections are strongly saturated for bright light sources.

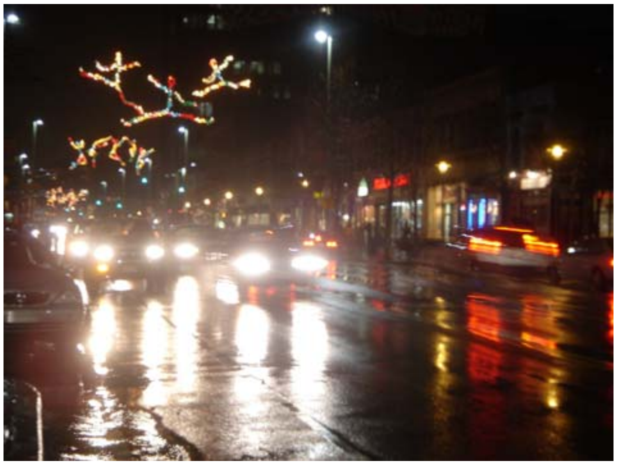


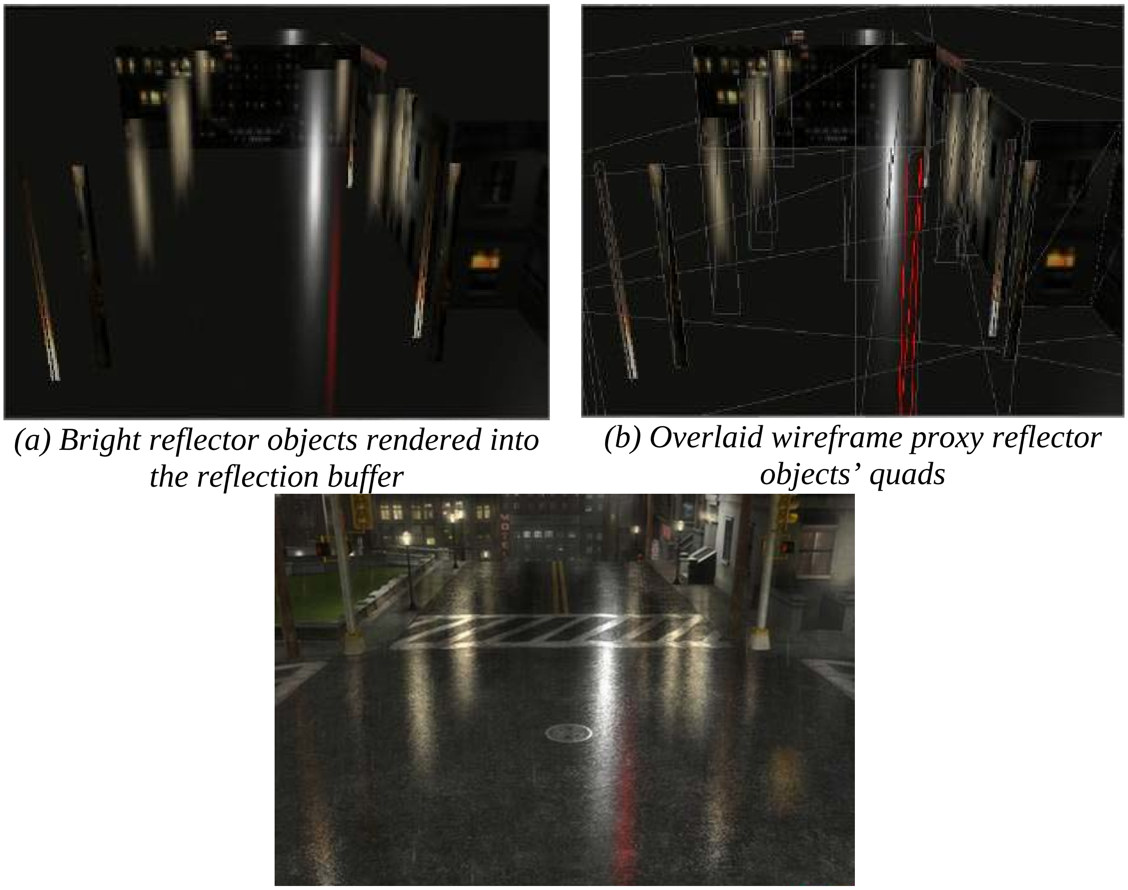
Figure 16. *Real-life photograph of a rainy night in Central Square, Cambridge, MA.*

A good example of real-life scene during the rain in Central Square in Cambridge is in Figure 16 on the right. There we see a number of store signs, car head and tail lights and street lights reflected in the street. Notice that the original shape of each reflector is only distinguishable by the blurred dominant colors (such as the reddish-orange glow of the taxi tail lights or the nearly white blobs of cars headlight reflections. Similarly, we want to preserve the brightest principal colors and create a blurry glowing reflection image for each light source or bright reflecting object in our scene.

Realistic streaky reflections increase the feel of rain on wet streets and surfaces. In our environment we create reflections for all bright objects onto the paved streets and large flat surfaces, such as the rooftop ledge (see figure 18 for examples of reflections in our interactive scene). All objects that can be viewed as reflectors are identified as such by the artists a priori. Examples of the bright reflector objects in our environment are the neon lights (such as the toy shop sign, street and building lamps (such as the lamp on the rooftop), the car head and tail lights, and bright building windows). Note that we render both bright *light* objects (such as street lamps), as well as the dark objects (such as the telephone poles and wires) (their colors are deepened).

Rather than simply rendering these objects directly into the reflection buffer as they are in the final rendering pass, we improve performance by rendering proxy geometry into the reflection buffer instead. For each reflector object the artists generate a quad with a texture representing the object, which is slightly blurred out (since these reflections tend to be blurry in the end) (see Figure 17a). Note that we aren't simply rendering the unlit proxy object texture into the reflection buffer. At run-time this proxy object is lit in a similar fashion to the original object (to make sure that the reflections appear to respond correctly to the environment lighting) during rendering into the reflection buffer. The reflection shader uses a simplified lighting model to preserve dominant colors, but does not waste performance on subtle effects of a particular material. This dynamic lighting allows us to represent reflected animated light sources (such as the flickering neon sign of the shop or the blinking traffic lights on the streets) correctly in the street reflections (which dim and light in sync with their corresponding reflector objects).

The proxy reflection objects are dynamically stretched view-dependently toward the viewer in the vertex shader (you can see the reflection quad objects with wireframe displayed in Figures 17a and 17b). The amount of stretching varies depending on the distance of the object to the viewer.



(a) Bright reflector objects rendered into the reflection buffer

(b) Overlaid wireframe proxy reflector objects' quads

(c) Resulting scene using the above reflection buffer after processing

Figure 17. *View-dependent streaky reflection rendering*

One aspect that we want to mention for using the proxy objects is the issue of culling the objects if the original reflector objects are no longer in the view. Since the proxy objects are only rendered into the offscreen reflection buffer, they do not go through the visibility culling process in our rendering engine. Therefore, we ran into a situation where the taxi cab, turning around a corner, would disappear from the view, but even a few seconds later we could still notice the stretchy red tail light reflections. To work around this problem, we place separate reflector blocker objects, which act to hide the proxy reflector objects from rendering into the reflection objects if the original reflector object is no longer in view.

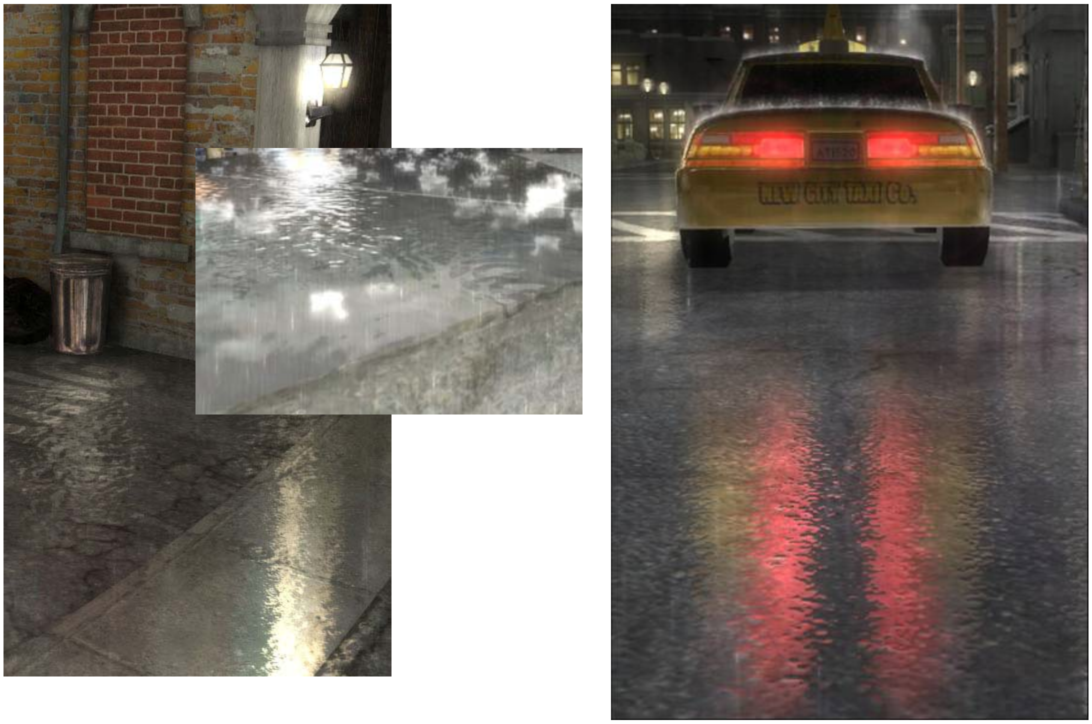


Figure 18. *View-dependent streaky reflections in the ToyShop demo*

For performance reasons the reflection buffer is scaled to be half size of the original back buffer (and a separate quarter sized reflection buffer for the rooftop reflections). We utilize an expanded dynamic range for representing the rendered colors so that we can preserve the brightest and darkest colors for reflections (such as street lamps or taxi headlights or telephone poles) by using the 10-10-10-2 HDR format for the reflection buffer.

Next we need to address the issue of making these view-dependent reflections appear blurry, glowing and streaky. For that, we turn to the post-processing system already in place (as described in section 3.3.1). We use a post-processing technique to dynamically streak the reflection buffer in the *vertical* direction only to simulate warping due to raindrops striking in the puddles. Note that this is done in separate passes from the regular scene post-processing.

During rendering of the final scene prior to post-processing, we sample from the reflection buffer using screen space projection of the input vertex coordinate for each reflective material (such as the street pavement or the roof ledge granite, see Figure 18). Reflections are also distorted based on the normals of the surface they pass through. We use object’s per-pixel normal in tangent space to account for stretching of the reflection in view space and warp the reflection based on the surface normal. The post-process-based blurring and this warping aid in removing specular aliasing and excessive flickering from reflections which would otherwise be highly distracting.

Since the number of draw calls in an interactive rendering application is an example of a typical rendering bottleneck of many games, we paid particular care to their optimization. Given the sheer number of various effects we designed to implement in our environment, we had very strict requirements for performance tuning and tried to save every percent of the frame rendering time. Since we rendered a large number of reflector objects, the goal was to render them in a single draw call. This was accomplished by specifying their world position in the skinning matrix using only a single bone. Therefore all objects with similar materials (such as the telephone poles or the street lamps) were rendered as one single big object using skinning to position them around the scene.

3.5.2 Dynamic reflections for a reflective taxi

While the taxi cab is moving through the streets of our interactive city, the environment is reflected in its metallic and glass surfaces (Figure 16a). We implement these reflections through the environment map reflection method (see [Akenine-Möller02], pages 153-166 for more details). In order to generate dynamic reflections, we render our environment into a cubemap with the camera placed at the center of the taxi cab as its moving through the scene. This dynamic cubemap is used for reflection color lookup for the cab surfaces (Figure 19 below shows an example of the contents of this cubemap).

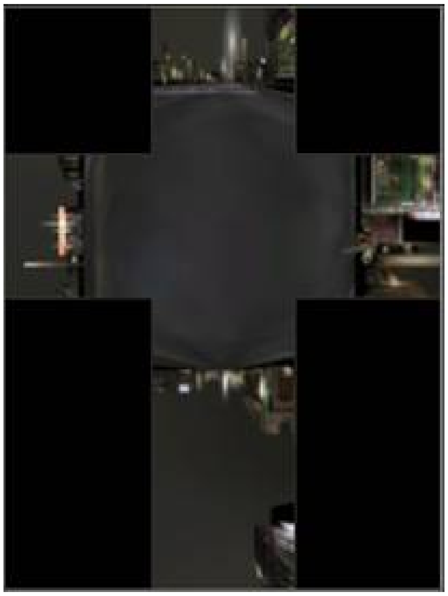


Figure 19. *An example of the dynamically rendered environment map for taxi cab reflections.*

Using the rendering script allows us to only render the environment cubemap for frames when the taxi was actually moving. At the same time, note that the environment gets rendered 6 times (for every face of the cubemap), so rendering the full scene is suboptimal. To improve that, we build a low resolution ‘billboard’ version of the city environment. We place the billboard quads along the taxi cab path. The quads contain textures of the buildings and environment as viewed from the point of view of the cab. These textures are created by taking in-engine snapshots by placing the camera on the taxi cab path while rendering the full scene (figure 20 contains an example of this billboard texture atlas). Similar to the approach for rendering reflector objects in section 3.5.1, we light these quads dynamically to get more accurate reflections in the final rendering. However, rendering just the billboard quads into the faces of the cubemap (rather than the full geometry) saves a great deal on performance. Instead of using the billboard versions of the environment, another suggestion for in-game rendering would be to use one of the lower levels of details of the scene, if the game contains support for level-of-detail rendering.

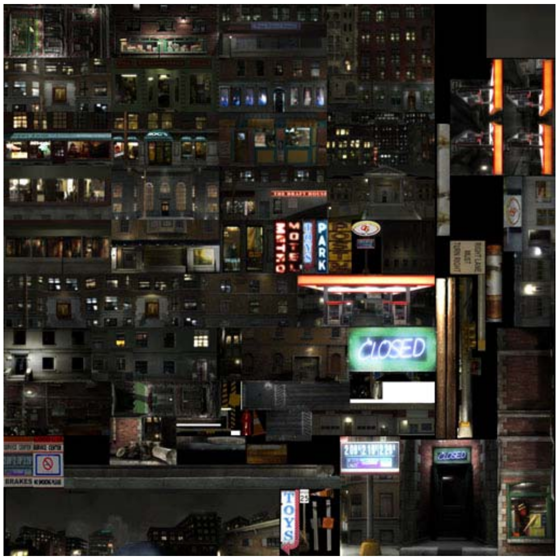


Figure 20. *Billboard dynamic reflector texture atlas*

3.6 Rendering rain

Rain is a complex visual phenomenon. It is composed of many visual components. Rainfall consists of specially distributed water drops falling at high velocity. Each individual drop refracts and reflects the environment. As the raindrops fall through the environment, they create the perception of motion blur and generate ripples and splashes in the puddles. Rain effects have been extensively examined in the context of atmospheric sciences ([Wang75] and [Mason75]), as well as in the field of computer vision ([Garg04]). We developed a number of effects for rendering rain in our interactive environment in real time, consisting of a compositing effect to add rainfall into the final rendering, a number of particle-based effects and dynamic water effects, simulated directly on the GPU.

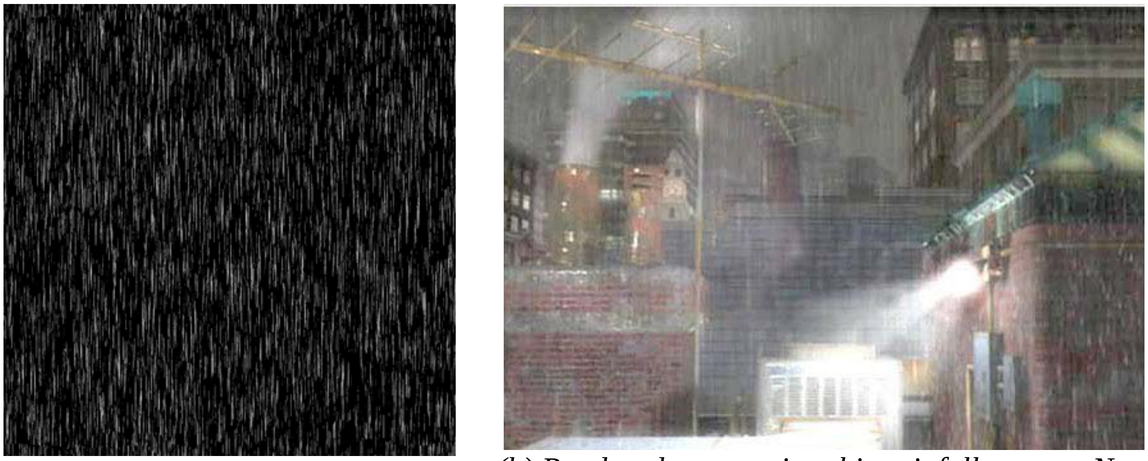
3.6.1 Rendering multiple layers of rain with a post-processing composite effect

We developed a novel post-processing rain effect simulating multiple layers of falling raindrops in a single compositing pass over the rendered scene. We create motion parallax for raindrops utilizing projective texture reads. The illumination for rain is computed using water-air refraction for individual raindrops as well as reflection due to surrounding light sources and the Fresnel effect. We provide a set of artist knobs for controlling rain direction and velocity, and the rainfall strength. The raindrop rendering receives dynamically-updated parameters such as lightning brightness and direction from the lightning system to allow correct illumination resulting from lightning strikes.

Creating rainfall We render a composite layer of falling rain as a full-screen pass before the final post-processing of the scene. Rainfall is simulated with an 8 bit texture (see

Figure 20 for an example texture and the resulting rain scene). To simulate the strong mistiness of the raindrops, we blur the rain by using the post-processing system (as described in section 3.3.1). The artists can specify the rain direction and speed in world-space to simulate varied rainfall strength.

Although so far this approach sounds rather straight-forward, there are several challenges with rendering rain through a composite layer. The first difficulty lies in minimizing repeating patters that are inevitable when using a single static texture to model dynamic textured patterns. The second concern lies with the consideration that the rain pass is a full-screen pass, and therefore every pixel on the screen will go through this shader. This has direct effect on performance, and we must design the composite rain rendering such that it gives pleasing visual results without an expensive shader.



(a) Rainfall texture

(b) Rendered scene using this rainfall texture. Note that the image intensities have been brightened for better contrast since this is a static capture of rain.

Figure 20. *Rainfall texture applied for a composite rain effect in the interactive scene*

Computer vision analysis of rain models ([Garg04]) and video rain synthesis ([Starik03]) helps us to observe that one cannot easily recognize rainfall from a single static frame; however, rain is easily noticeable in a dynamic simulation or a video. Perceptual analysis of rain video shows that the individual raindrop motion cannot be tracked by human perception accurately due to swift movement and density of raindrops, which allows us to assume temporal independence of rain frames. However, our empiric experiments showed that purely random movement of raindrops does not yield satisfactory results (generating excessive visual noise). Therefore to simulate strong rainfall, we simultaneously use the concepts of individual rain drop rendering and the principles stochastic distribution for simulation of dynamic textures (as in [Bar-Joseph01] and [Doretto03]).

The first part of our algorithm simulates individual rainfall movement. The artist-specified rain direction vector is moved into clip space. We use this vector to determine a raindrop position in screen space by using the current position in clip space, specified rainfall velocity and current time. Given these parameters and computed the raindrop position, we can scroll the rainfall texture using the specified velocity vector. However, although texture scrolling is a very straight-forward approach, even with several texture fetches in varied directions with slight randomization, repeating rain patterns become rather obvious in a full-screen pass.

Multiple layers of rain Our goal is to simulate several layers of raindrops moving with different speeds at varied depths in a single rendering layer. This better approximates real-life rain movement and allows us to create a feeling of raindrop motion parallax (a strong visual cue in any dynamic environment). The artists can specify a rain parallax parameter which provides control for specifying the depth range for the rain layers in our scene. Using the concepts of stochastic distribution for simulation of dynamic textures, we compute a randomized value for an individual raindrop representation to use in the rain shader. Using the rain parallax value, the screen-space individual raindrop parameter and the distribution parameter, we can model the multiple layers of rain in a single pass with a single texture fetch. This allows us to simulate raindrops falling with different speed at different layers. The rain parallax value for the rain drop, multiplied by a distribution value, can be used as the *w* parameter for a projective texture fetch to sample from the rainfall texture. Note that we use a single directional vector for all of our raindrops which is crucial for creating a consistent rainfall effect. This creates excellent visual effects of random streaking for the raindrops.

Rain appearance Given a moving raindrop, we need to shade it. Raindrops behave like lenses, refracting and reflecting scene radiances towards the camera. They refract light from a large solid angle of the environment (including the sky) towards the camera. Specular and internal reflections further add to the brightness of the drop. Thus, a drop tends to be much brighter than its background (the portion of the scene it occludes). The solid angle of the background occluded by a drop is far less than the total field of view of the drop itself. In spite of being transparent, the average brightness within a stationary drop (without motion-blur) does not depend strongly on its background.

Falling raindrops produce motion-blurred intensities due to the finite integration time of a camera. Unlike a stationary drop, the intensities of a rain streak depend on the brightness of the (stationary) drop as well as the background scene radiances and integration time of the camera. We simulate the motion blur for the raindrops by applying blurring via post-processing after the rain pass has been blended onto the scene rendering. This simulates both raindrop motion-blur and multiple-scattering glow for individual raindrops. To shade an individual raindrop, we use a tangent-space normal map corresponding to the rainfall texture. Note that since this is a full-space pass, the tangent space is simply specified by the view matrix. For each pixel in the rain pass, we compute reflection based on the individual raindrop normal and air-to-water refraction. Both are attenuated toward the edges of the raindrop by using the Fresnel effect.

Raindrop transparency An interesting observation is that as the lightning strikes, the raindrops should appear more transparent. In other words, the opacity of each individual raindrop must be a function of the lightning brightness; otherwise water surfaces appear too solid. As mentioned in section 3.3, our rendering script propagates the lightning system parameters to all of our rain shaders, as well as the material shaders. For the raindrop rendering, we use a combined lightning brightness parameter (mixing both lightning ‘light sources’ as they flash in the environment) to compute the bias value to adjust the amount of reflection and refraction.

Realistic rain is very faint in bright regions but tends to appear stronger when the light falls in a dark area. If this is modeled exactly, the rain appears too dim and unnoticeable in many regions of the scene. While this may be physically accurate, it doesn't create a perception of strong rainfall. Instead of rendering a precise representation, we simulate a Hollywood film trick for cinematic rain sequences. The film crew adds milk to water or simply films milk to make the rain appear stronger and brighter on film. We can bias the computed rain drop color and opacity toward the white spectrum. Although this may seem exaggerated, it creates a perception of stronger rainfall

Compositing rain via blending We would like to make a few notes on specifying the blending for the rain pass. The rain layer is rendered both as a transparent object and a glowing object (for further post-processing). However, since we wish to render the rain layer in a single pass, we are constrained to using a single alpha value. Controlling both opacity and glow with a single alpha blending setting can be rather difficult. Despite that, we want to render transparent objects that glow, controlling each state separately for better visual results. We found that we can use two sets blending parameters to control blending for glow and for transparency for all rain effects all rain effects (composite rain, raindrops, splashes). In the latest DirectX9.0c there is a rendering state for separate alpha blending called `D3DRS_SEPARATEALPHABLENDENABLE`. Using this state along with the regular alpha blending function (via `D3DRS_ALPHATESTENABLE`) allows us to specify two separate blending functions for the regular opacity blending and for the alpha used for glow for post-processing blurring pass.

Finally, we would like to mention a few other considerations for including this composite post-processing rain layer effect in other interactive scenarios such as games. In many extensive environments which may include changing weather conditions as well as the changes between outdoor and indoor locations, the issue of controlling composite rain rendering can appear challenging. In reality it is not so – there is a number of ways to efficiently accomplish that goal. In our interactive scene, we use the rendering script to determine whether the camera is located inside the toy store or whether it is outside. This information is used to dynamically turn off composite rain rendering. A similar concept (an engine state specifying what environment the camera is located, for example) can be used in many game setups. Likewise, an engine state that specifies the current weather condition can be used to control rain rendering by turning on and off rendering of the rain quad. If there is no notion of the appropriate engine state, another approach may involve using a sky visibility overhead lightmap (see section 3.6.3 for more on overhead lightmap). One can encode a Boolean sky visibility value (precomputed for the entire environment at preprocessing time for every point in the scene, similar to the overhead lighting lightmap used in section 3.6.3 for raindrop splashes lighting). This value can be used directly in the rain quad pixel shader to turn off rendering pixels based on the current camera location. However, we would like to note that this approach is far less efficient than the rendering script-based control of rain rendering.

3.6.2 Raindrop Particles Rain

To simulate raindrops falling off various objects in our scene, we use screen-aligned billboard particle systems with normal-mapped rain droplets (Figure 22a and 22b). In our scenes we found that using on the order of 10-15,000 particles gives excellent results.

We created a base template particle system that uses the physical forces of gravity, wind and several artist-animated parameters. The artists placed a number of separate particle systems throughout the environment, to generate raindrops falling off various surfaces, such as rooftop ledge, building lamps, and so on, onto the streets.

To render an individual raindrop particle, we stretch the particle billboard based on the particle velocity (with slight randomization offsets to vary velocity per individual particle within a particle system). The illumination model used for these particles is similar to that of the composite rain layer. We use a normal map for a water droplet for each individual raindrop. Instead of using an accurate droplet-shape representation, we pre-blurred and stretched the drop normal map to improve the perception of motion blur as the raindrops move through the environment (Figure 21). Note that the tangent space for a billboard particle is defined by the view matrix.

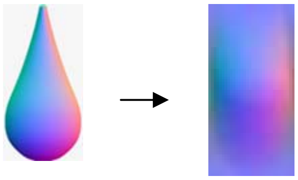
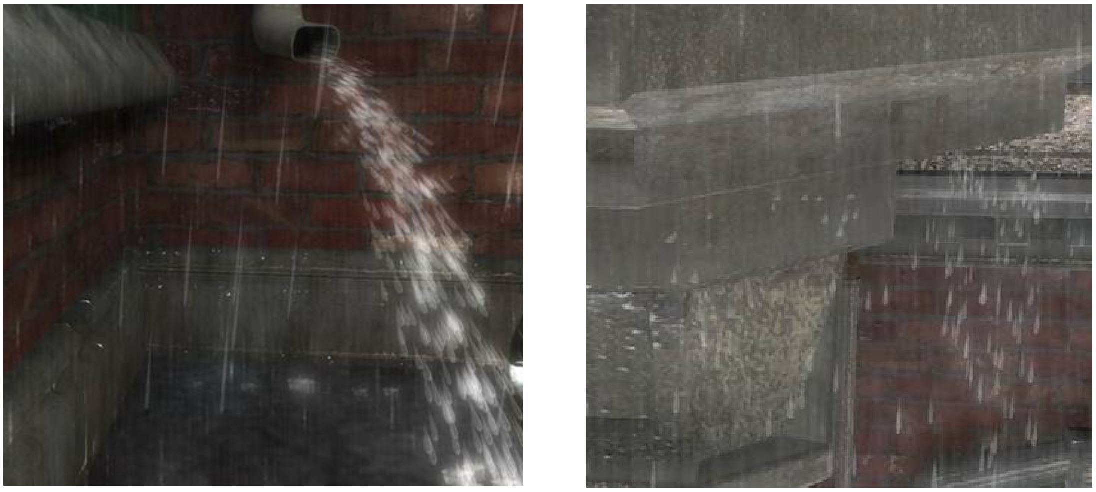


Figure 21*. Pre-blurred droplet normal map (on the right)*

To shade the raindrop particle, we only compute specular reflection and air-to-water refraction effects, using the pre-blurred normal map. Since droplet should appear more reflective and refractive when a lightning flashes, biased lightning brightness value adjusts the refraction and reflection color contributions.



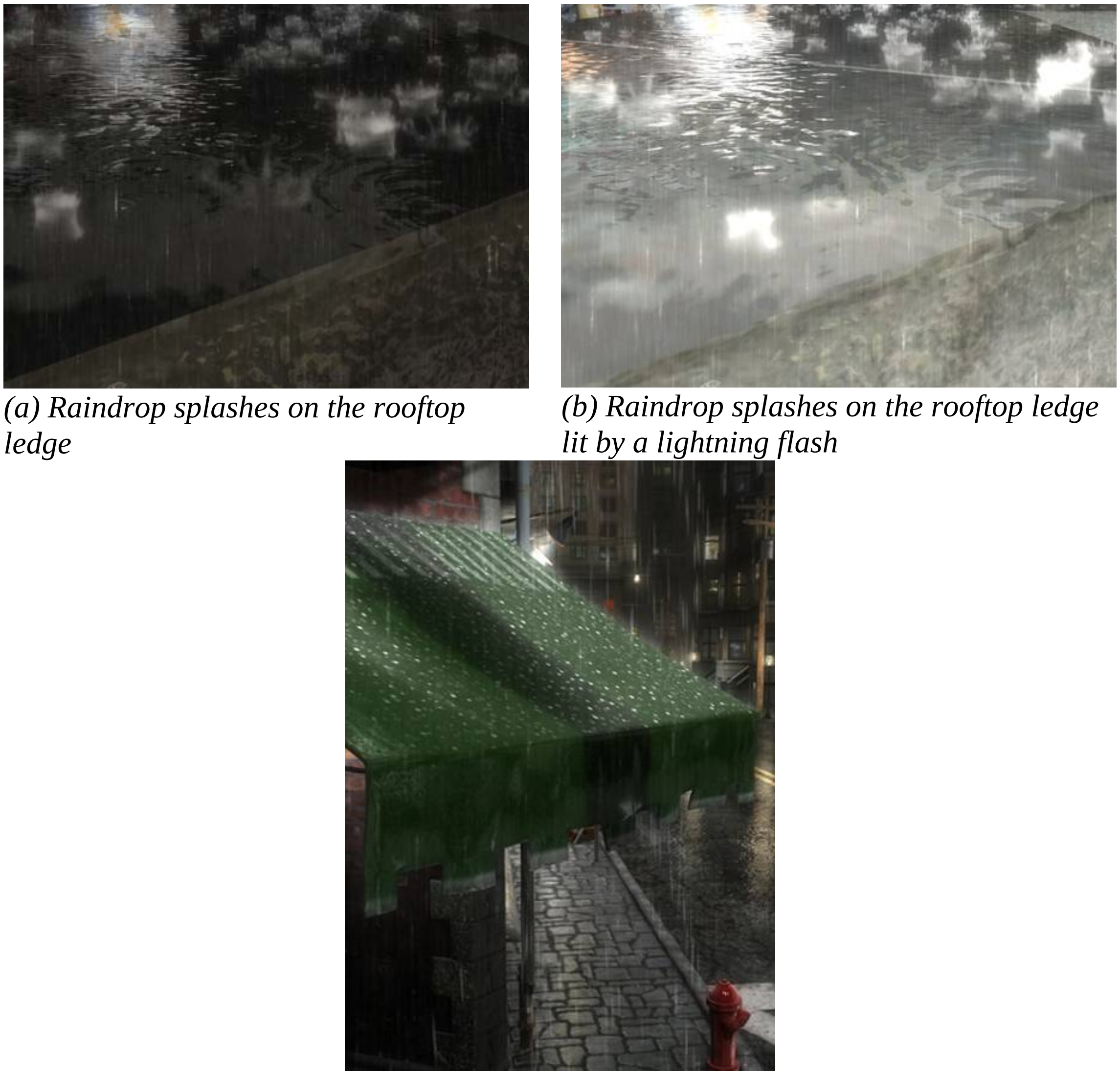
(a)Raindrops pouring from a gutter pipe *(b) Raindrops falling off the rooftop ledge*

Figure 22. *Raindrops falling off objects in our environment.*

To control raindrop transparency, we attenuate raindrop opacity by its distance in the scene. We wish to make the individual raindrop particles appear less solid and billboard-like as they move through the environment. This can be accomplished by attenuating the particle opacity value by Fresnel value, scaled and biased by two artist-specified parameters for droplet edge strength and bias (which could be specified per particle system). We used the observation that the raindrops should appear more transparent and water-like when the lightning strikes, and increased the raindrop transparency as a function of the lightning brightness to maintain physical illusion of water. This can be easily done by biasing droplet transparency by 1 – ½ * *lightning brightness*. The particles still maintain their artist-specified transparency in the regular lighting without any lightning flashes. We used this approach for both regular raindrop rendering and for raindrop splash rendering.

3.6.3 Rendering raindrop splashes

We simulate raindrops splashing when hitting solid objects by colliding individual particles with objects in the scene (Figure 23c). In our system we use special collider proxy objects. In a different engine environment this may be done by colliding particles directly with game objects. We used on the order of 5-8,000 particles to render raindrop splashes each frame. Figures 23a and 23b show an example of raindrop splashes rendered with regular illumination (a) and lit by a lightning flash on the rooftop ledge (a).



(a) Raindrop splashes on the rooftop ledge

(b) Raindrop splashes on the rooftop ledge lit by a lightning flash

(c) Raindrop splashing hitting the store awning

Figure 23. *Raindrop splash rendering*

To shade these splashes, we use a pre-rendered high-quality splash sequence for a milk drop (Figure 24). A single filmed high-quality splash sequence for a milk drop was used to drive the raindrop splash event for *all* of the thousands of splashing raindrops. The challenge lied in reducing the noticeable repetition of the splash animation (especially considering that viewer could get rather close to the splashes). To address this concern we incorporated a high degree of randomization for particle parameters (particle size and transparency), and dynamically flipped horizontal texture sampling for the filmed sequence based on a randomly assigned particle vertex color.

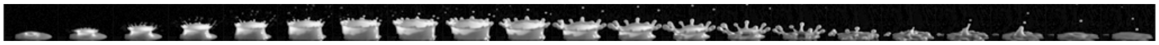


Figure 24. *Milk drop sequence for raindrop splash animation*

Splashes should appear correctly lit by the environment lights. We added backlighting to the splashes so that they accurately respond to the environment lights (and thus display the subtle effects of raindrops splashing under a street light). If light sources are behind the rain splashes, we render the splash particles as brightened backlit objects; otherwise we only use ambient and specular lighting for simplicity. We compute specular lighting for all available dynamic lights in the vertex shader for performance reasons.

Aside from the dynamic lights, we wanted to simulate the splashes lit by all of the bright objects in the environment (such as street lamps, for example), even though those objects are not actual light sources in our system. Using a special 'overhead' lightmap let us accomplish that goal (see Figure 25 for an example). We can encode the light from these pseudo light sources into a lightmap texture to simulate sky and street lamp lighting. We can then use the splash world-space position as coordinates to look up into this lightmap (with some scale and bias). The overhead lightmap value modulates otherwise computed splash illumination.

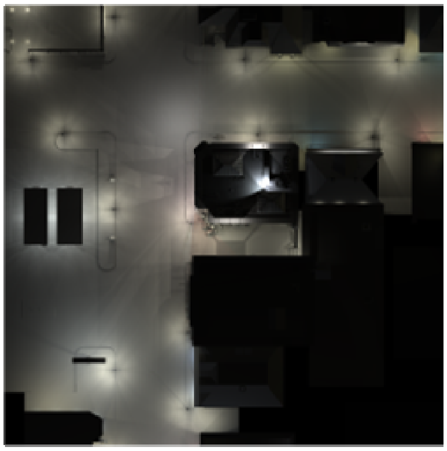


Figure 24. *Overhead lightmap example*

3.7 GPU-Based water simulation for dynamic puddle rendering

The raindrop particle collisions generate ripples in rain puddles in our scene. The goal was to generate dynamic realistic wave motion of interacting ripples over the water surface using the GPU for fast simulation. We use an implicit integration scheme to simulate fluid dynamics for rendering dynamically lit puddle ripples. Similar to real-life raindrops, in our system we generate multiple ripples from a single raindrop source which interact with other ripples on the water surface. The physics simulation for water movement is done entirely on the GPU. We treat the water surface as a thin elastic membrane, computing forces due to surface tension and displacing water sections based on the pressure exerted from the neighboring sections. Our system provides simple controls to the artists to specify water puddle placement and depth. Figure 25 shows water puddle on the rooftop and in the streets using our system.

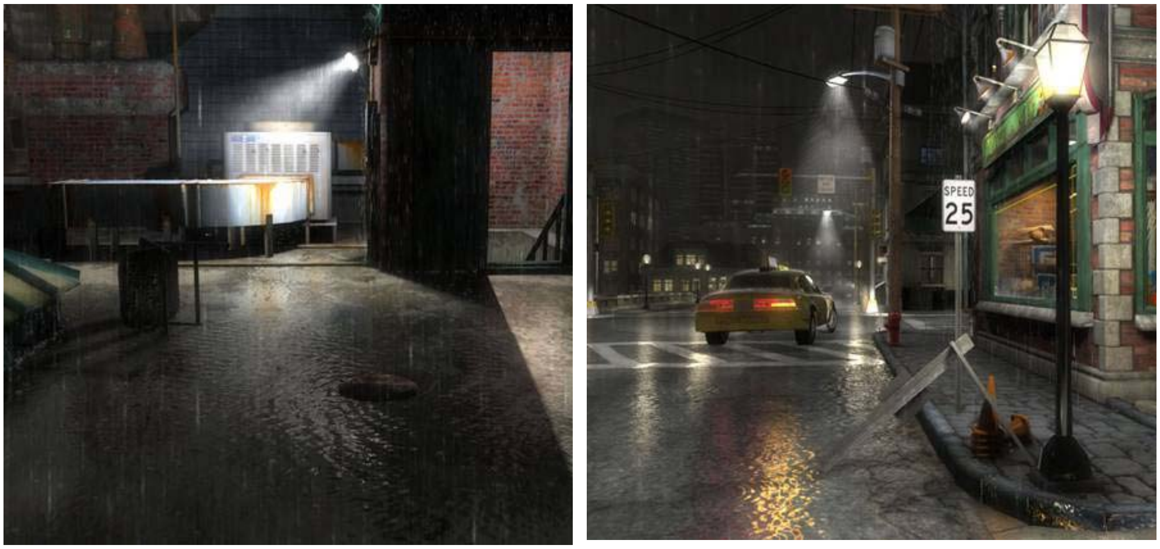


Figure 25. *Dynamic puddles with ripples from rain drops*

Water ripples are generated as a result of raindrops falling onto the geometry in the scene. This can be a direct response from the actual raindrop particle system colliding with the scene objects. In our implementation we approximated this effect by stochastically rendering raindrops into a 'wave seeding' texture. In the case of direct particle response, the approach is similar; however, the initial wave texture must be rather large to accommodate raindrops falling throughout the entire environment. In order to conserve memory, we decided against that approach, and limited our simulation to 256 x 256 lattice. Raindrop seeds are rendered as points into the water simulation texture, where the color of the raindrop is proportional to its mass. The method can be extended to generate dynamic water surface response for arbitrary objects. This can be achieved by rendering an orthographic projection of the objects into the seeding texture, encoding object's mass as the color of the object's outline. This would generate a wake effect in the water surface.

We render the raindrop seeds into the first water simulation buffer in the first pass. These rendered seeds act as the initial ripple positions. They 'excite' ripple propagation in the subsequent passes. In the next two passes we perform texture feedback approach for computing water surface displacements. In our case two passes are sufficient for the time step selected. If a larger time step is desired, more passes or a more robust integration scheme may be selected (we use Euler integration). In the fourth pass we use the Sobel filter ([Jain95]) on the final water displacement heights texture to generate water puddle normals.

Real-life raindrops generate multiple ripples that interact with other ripples on the water surface. We implement the same model. We render a raindrop into a wave seed texture using a dampened sine wave as the function for raindrop mass. This approximates the concentric circular ripples generated by a typical raindrop in the water puddle.

We approximate the water surface as a lattice of points. Each lattice point contains the information about the water surface in that location. In particular we store the current position as a height value and the previous time step position (see Figure 26). Since we perform all of the water simulation computations directly on the GPU, the lattice information is stored in a 32 bit floating point texture (with 16 bits per each position channel). A related approach was described in [Gomez00] where the lattice of water displacements was simulated with water mesh vertices displaced on the CPU.

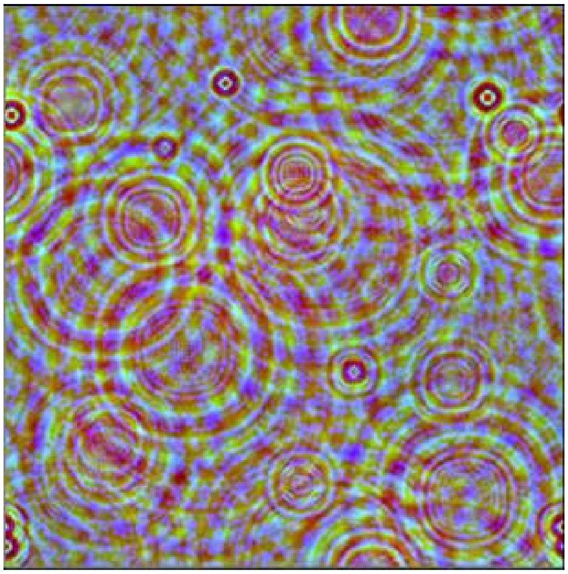


Figure 26. *Water displacements encoded into the feedback texture. The red channel contains water heights at current time step, and the green channel contains previous time step displacements.*

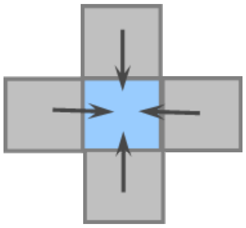


Figure 27. *Water neighbor cell acting on the current cell*

To compute water surface response we treat the water surface as a thin elastic membrane. This allows us to ignore gravity and other forces, and just account for the force due surface tension. At every time step, infinitesimal sections of the water surface are displaced due to tension exerted from their direct neighbors acting as spring forces to minimize space between them (Figure 27).

Vertical height of each water surface point can be computed with partial differential equation:

$$\frac{\partial^2 z}{\partial t^2} = v^2 \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right)$$

where

z is the water displacement height,
v is the velocity of the water cell
x and y are the lattice coordinates of the water cell

This PDE is solved with Euler integration in DirectX9.0 pixel shaders in real-time by using the texture feedback approach to determine water wave heights for each point on the lattice.

Water puddles integration. We render a single 256 x 256 water simulation for the entire environment. Therefore we have to use a bit of cleverness when sampling from this simulation texture - since many different objects all use the same wave ripples simulation at the same time. We sample from the water membrane simulation using the object’s current position in world space, specifically the *xz* coordinates as a lookup texture coordinates into the wave normal map (Figure 28).

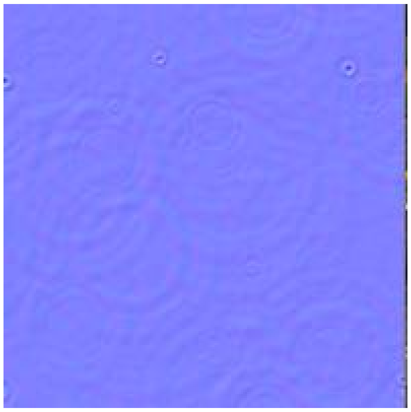


Figure 28. *Dynamic ripples normals*

The artists control the sampling space per object with a scaling parameter that allows them to scale the size of water ripples at the same time (by essentially scaling the lookup coordinates). To reduce visual repetition for puddles, we rotate the water normals lookup coordinates by an angle specified per-object. Since we sample from the water normals texture when rendering an object with puddle, we do not require additional puddle geometry. It is even possible to dynamically turn water puddle rendering on and off by simply using a shader parameter and dynamic flow control. To render an object with water puddles, we perturb the original object's normal from a bump map with the normal from the water membrane simulation.

The artists can also specify a puddle ripple influence parameter per object. This parameter controls how much the water ripple normal perturbs the original bump map normal. This allows us create different water motion for various objects.

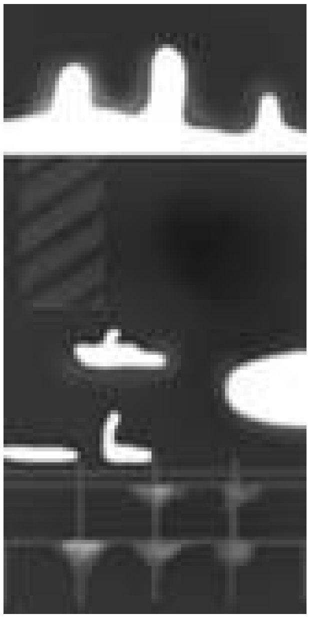


Figure 29. *Puddle depth and placement map.*

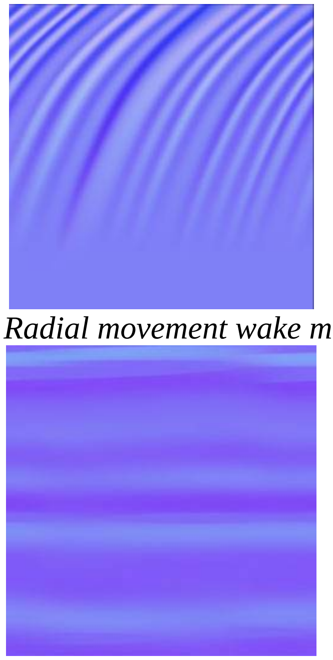
Puddle Placement and Depth To render deep puddles, we use just the water puddle normal sampled as just described, along with the color and albedo attributes of the object. We wanted to mimic varied puddle depths of the real-world and allow artists creative control over the puddle placement. A puddle depth mask was our answer (Figure 29 on the left). Adding puddles with ripples to objects is straight-forward:

- Define the ripple scale parameter and sample ripple normals using the world-space position

- Sample puddle depth map

- Interpolate between the object normal map and the water ripple surface normal based on the puddle depth value and artist-specified puddle influence parameter

Creating Swirling Water Puddle For the rooftop puddle, we want to create an impression of water, swiftly swirling toward the drain, with ripples from raindrops warping the surface (using the above approach). We used several wake normal maps to create the whirlpool motion. The first normal map (Figure 30a) was used to swirl water radially around the drain. Combined with it, we used the wake normal map from Figure 30b to create concentric circles toward the drain.



(a) Radial movement wake map

(b) Concentric circles wake map

Figure 30. *Wake normal maps*

3.8 Raindrop movement and rendering on glass surfaces in real time

We adopted the offline raindrop simulation system [Kaneda99] to the GPU to convincingly simulate and render water droplets trickling down on glass planes in real-time. This system allows us to simulate the quasi-random meandering of raindrops due to surface tension and the wetting of the glass surfaces due to water trails left by droplets passing on the window. Our system produces a correctly lit appearance including refraction and reflection effects.

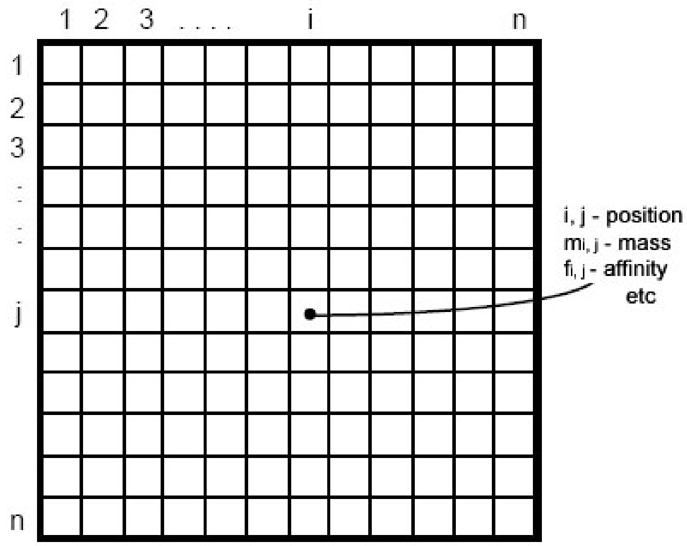


***Figure 31**. Discrete lattice model for storing water droplet information at run-time*

Droplet movement. The glass surface is represented by a lattice of cells (Figure 31) where each cell stores the mass of water in that location, water x and y velocity, and the amount of droplet traversal within the cell. This information can be conveniently packed into a 16 bit per channel RGBα texture. Additionally we store droplet mass and affinity information for each cell as well.

The force of gravity depending on the mass of the droplet is used to compute the downward movement force on the droplet. Static friction for stationary droplets and dynamic friction for moving droplets is used to compute the competing upward force. The static and dynamic friction varies over the surface of the glass. This resultant force is applied to the initial velocity to determine the new velocity value for the droplet.

At any given time, droplets can flow into the three cells below its current cell. New cell for the flow is randomly chosen, biased by the droplet directional velocity components, friction based affinity of current cell and the 'wetness' of the target cell. The glass friction coefficients are specified with a special texture map. Droplets have a greater affinity for wet regions of the surface. We update the droplet velocity based on the selected cell.

Droplet rendering. First we render the background scene. Then we render the water droplet simulation on the window. This allows us to reflect and refract the scene through the individual water droplets. In order to do that, we use the water density for a given rendered pixels. If there is no water in a given pixel, we simply render the scene with regular properties. However, if the water is present, then we can use the water mass as an offset to refract through that water droplet. At the end of the droplet movement simulation, each cell contains a new mass value (Figure 32). Based on the mass values, we can dynamically derive a normal map for the water droplets. These normals are used to perturb the rendered scene to simulate reflection and refraction through water droplets on the glass surface (figure 33a).

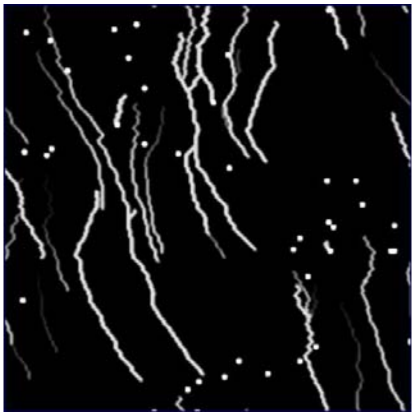
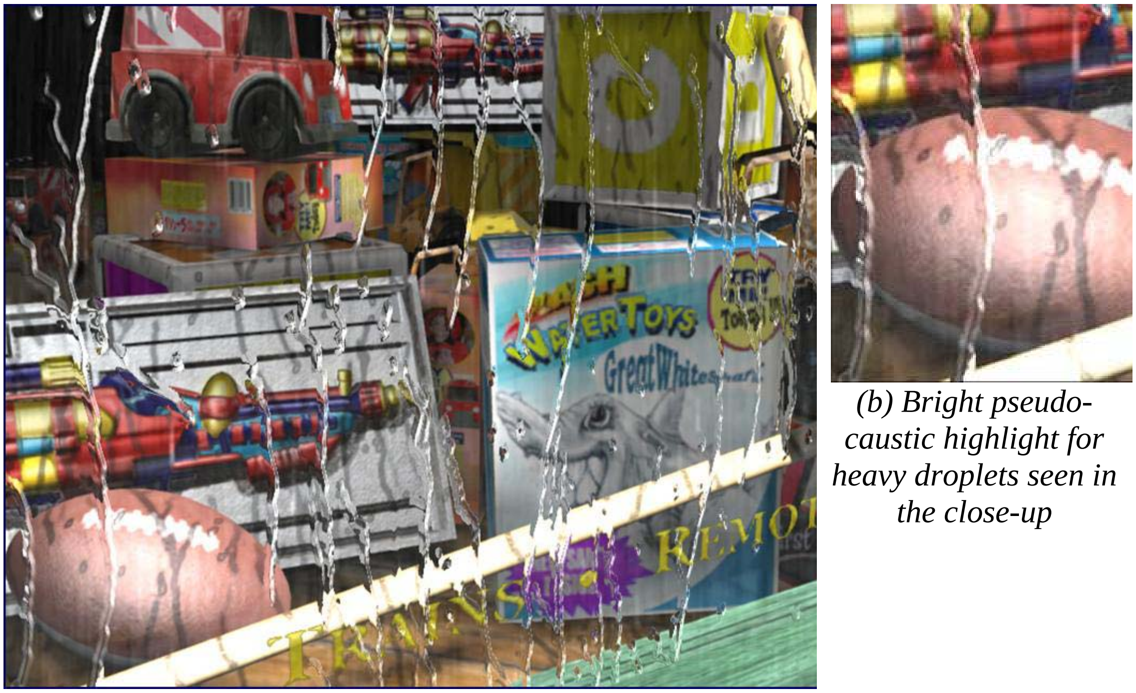


Figure 32. *Water droplet mass*

The droplet mass is also used to render dynamic shadows of the simulation onto the objects in the toy store (using the mass texture as a projective shadow for the other objects). If the droplet mass is large enough, we render a pseudo-caustic highlight in the middle of the shadow for that droplet (Figure 33b)



(b) Bright pseudo-caustic highlight for heavy droplets seen in the close-up

(a) Water droplet rendering on the glass window

Figure 33. *Water droplets refracting the scene of the toy store interior through and reflecting external lights. Note the shadows from the water droplets on the toys inside*

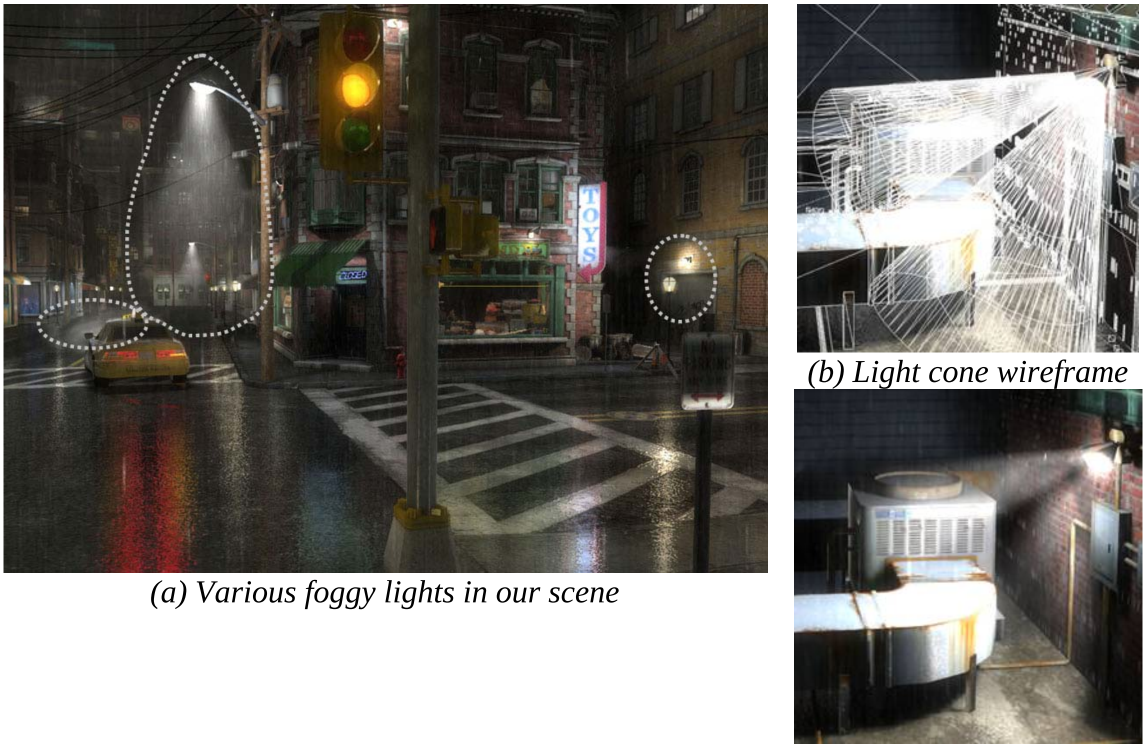
3.8 Effects medley

Aside from the main rain-related effects, we developed a number of secondary objects' effects that we would like to briefly mention here since they help increase the realism of our final environment.

3.8.1 Foggy lights in the street

Our scene has many foggy lights with rain (Figure 34a). We used an approximation shader instead of an expensive volumetric technique to render the volume of light under each lamp. We can simulate these lights as pseudo-volumetric light cones by rendering noisy cloud-like fog on the light cone surface (light cone objects' wireframes are shown in Figure 34b). This efficient approach uses a tangent space technique to control lit fog fading towards the edges of the light frustum (Figure 34c). In order to get smooth falloff on the silhouette edge of the cone, the view vector is transformed into tangent space and its angle is then used to attenuate the falloff. In the pixel shader we simulate distance attenuation of a light by a square of the *v* texture coordinate. Then we perform two fetches from a noise map scrolled in different directions to create perception of participating media in this light's frustum.

To render rain, falling under the bright light, we render an inserted quad plane in the middle of the light frustum with rain texture scrolling vertically matching the composite rain effect. The rain must fade out toward the edges of the frustum in the same fashion as the volumetric light. We use a map to match the cone light attenuation. For angled lamps, we orient the inserted rain quad around the up-axis in world-space in the vertex shader to ensure that the rain continues to respect the laws of gravity and falls downwards.



(a) Various foggy lights in our scene

(b) Light cone wireframe

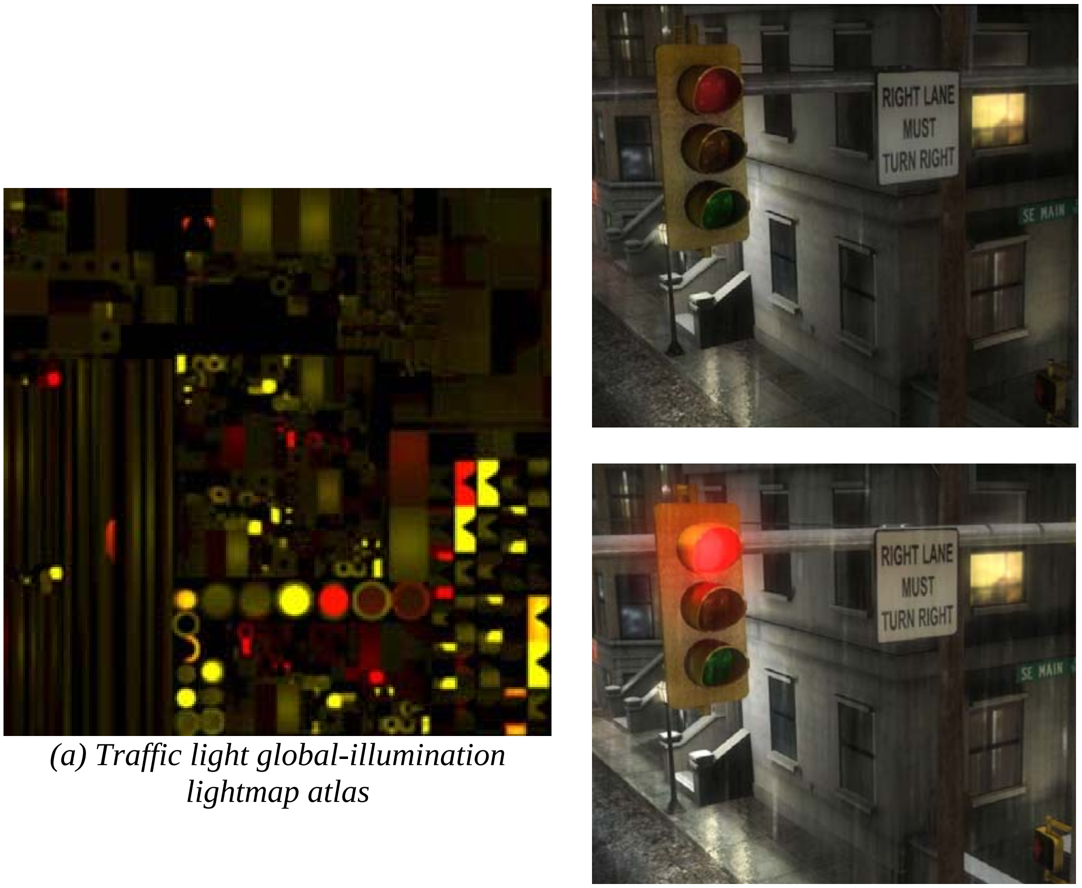
(c) Rooftop light cones

Figure 34. *Foggy volumetric light rendering*

3.8.3 Traffic light illumination

The traffic lamps should dynamically illuminate the traffic light signal object. Computing the full global illumination effects to simulate color bleeding and inter-reflection is an expensive operation (see chapter 9 for a longer discussion of global illumination effects). As a different approach, we use the concept of lightmap to help us simulate these subtle effects in a more efficient manner at run-time. We precompute the global illumination lightmap for the traffic light object, animated in accordance with the blinking traffic light signal (Figure 35a). This lightmap stores color as a result of color bleeding and inter-reflection effects computed with Maya®'s Mental Ray. Note that for a compact object such as the traffic light, this lightmap is very small. At rendering time, we sample the color of the lightmap using the traffic lamp animation parameter as a parameter for computing sampling texture coordinates.

We simulate the internal reflection through the colored reflective glass of the traffic lamp by fetching a normal from the glass normal map and using that normal vector to look-up into an environment map, coloring the resulting reflection color by the lamp color. These internal reflections will only appear when the lamp's glow is faint. We reflect the other parts of the lamp on the outside surface of the glass by another environment map fetch to render external reflections.



(a) Traffic light global-illumination lightmap atlas

(b) Traffic light in off (above) and on (below) state

Figure 35. *Approximating color bleeding for traffic light illumination*

3.8.4 Rendering Misty Rain Halos on Objects

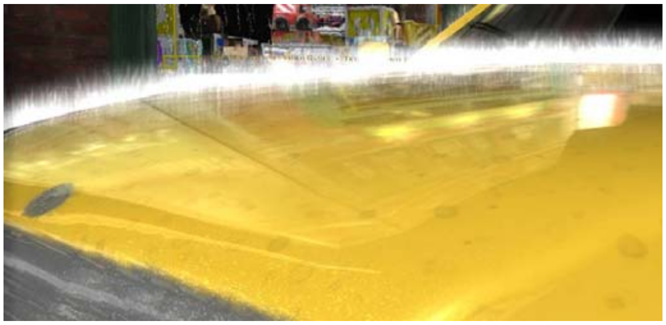
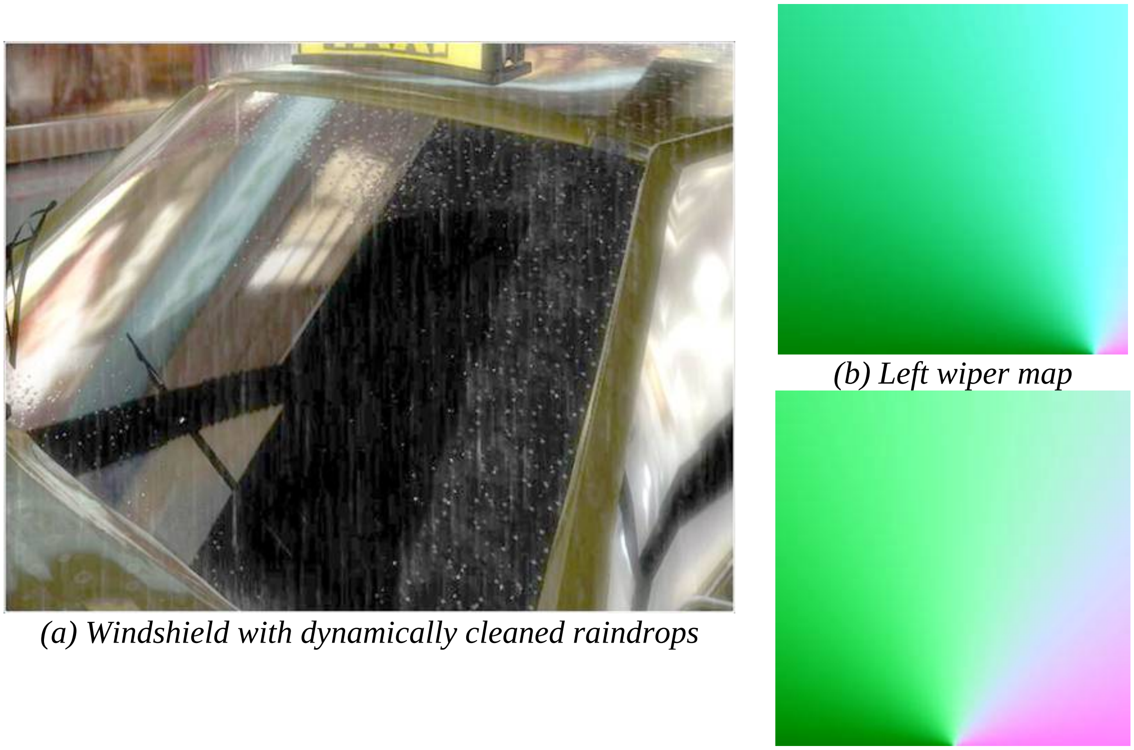


Figure 36. *Misty halos on the taxi with a fins effect and rain splatter via a shells effect*

In a strong rainfall, as the raindrops strike solid objects, they generate not only the splashes, but also a misty halo outlines along the edges of objects. We created a similar effect using the fins and shells technique (similar to real-time fur rendering from [Isidoro02]) (Figure 36). The rain halos are rendered with fin quads with scrolling rain (similar to the composite rain effect). Note that this effect requires additional fin geometry. Using the shells approach, we render rain splatters on the surface of objects in the form of concentric circles. In each successive shell we expand the splash circle footprint with a series of animated texture fetches and blend onto the previous shells.

3.8.5 Taxi windshield wipers effect for wiping off the droplets

The taxi cab windshield wipers can dynamically wipe away the static raindrops on the windshield (Figure 37a). Computing collision with the wipers and affecting droplet movement as a function of that calculation was not practical in our scenario due to many other effects already in place. Since the windshield wasn't prominent in the main fly path through our environment, we wanted an inexpensive approach to render this effect. As a solution, we use two wiper maps (Figures 37b and 37c) to determine which regions on the windshield were recently swiped clean by the wipers. The animation parameters from the wipers are used in the shaders in conjunction with the wiper maps to control the rendering of raindrops depending on which regions were wiped. We use two separate maps so that the wiped regions can overlap, similar to the real-life windshield wipers.



(a) Windshield with dynamically cleaned raindrops

(b) Left wiper map

(c) Right wiper map

Figure 37. *Taxi windshield rendering with animated windshield wipers*

3.9 Conclusions

Rain is a very complex phenomenon and in this chapter we presented a number of effects that help to generate an extensive, detail-rich urban environment in stormy weather. Each technology applied to the ToyShop demo adds detail to the scene. Each additional detail changes the way we experience the environment. It is this attention to detail that seduces the viewer and delivers a lasting impression of the limitless expression of real time graphics. All of these combined effects allow us to create a very believable, realistic impression of a rainy night in a cityscape at highly interactive rates. Rich, complex environments demand convincing details. We hope that the new technology developed for this interactive environment can be successfully used in the next generation of games and real-time rendering.

3.10 Acknowledgements

We would like to thank the ATI ToyShop team whose hard work and dedication resulted in the striking images of this interactive environment. The artists: Dan Roeger, Daniel

Szecket, Abe Wiley and Eli Turner, the programmers: John Isidoro (who has been a crucial part of the development of many effects described in this chapter and to whom we are deeply thankful for his insightful ideas), Daniel Ginsburg, Thorsten Scheuermann, Chris Oat, David Gosselin, and the producers (Lisa Close and Callan McInally). Additionally, we want to thank Jason L. Mitchell from Valve Software and Chris Oat for their help reviewing this chapter and overall encouragement and good humor.

3.11 Bibliography

AKENINE-MÖLLER, T., HEINES, E. 2002. *Real-Time Rendering*, 2nd Edition, A.K. Peters

BAR-JOSEPH, Z., EL-YANIV, R., LISCHINSKI, D., AND M. WERMAN. 2001. Texture mixing and texture movie synthesis using statistical learning. IEEE Transactions on Visualization and Computer Graphics, 7(2):120-135.

DORETTO, G., CHIUSO, A., WU, Y. N., SOATTO, S. 2003. Dynamic textures. International Journal of Computer Vision, 51(2):91-109.

HDRSHOP: HIGH DYNAMIC RANGE IMAGE PROCESSING AND MANIPULATION, version 2.0. Available from http://www.hdrshop.com/

GARG, K., NAYAR, S. K., 2004. Detection and Removal of Rain from Videos. IEEE Conference on Computer Vision and Pattern Recognition, pp. 528-535

GOMEZ, M. 2000. Interactive Simulation of Water Surfaces, Game Programming Gems. De Loura, Marc (Ed.), Charles River Media.

ISIDORO, J. 2006. Shadow Mapping Tricks and Techniques. In the proceedings of Game Developer Conference, San Jose, CA https://www.cmpevents.com/sessions/GD/S1616i1.ppt

JAIN, R., KASTURI, R., SCHUNK, B. G. 1995. Machine Vision. McGraw-Hill.

KANEDA K., IKEDA S., YAMASHITA H. 1999 Animation of Water Droplets Moving Down a Surface, Journal of Visualization and Computer Animation, pp. 15-26

KAWASE, M. 2003. Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L (Wreckless), GDC 2003 lecture. San Jose, CA.

MASON, B. J. 1975. Clouds, Rain and Rainmaking. Cambridge Press.

ISIDORO, J. 2002. User Customizable Real-Time Fur. SIGGRAPH 2002 Technical sketch.

NARASIMHAN, S.G. AND NAYAR, S.K., Shedding Light on the Weather. IEEE CVPR, 2003.

PERSSON E. 2006. HDR Texturing. ATI Technologies Technical Report, ATI SDK, March 2006. http://www2.ati.com/misc/sdk/ati_sdk(mar2006).exe

REINHARD E., WARD G., PATTANAIK S., DEBEVEC P., 2005, High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting. Morgan Kaufmann

REEVES, W. T., SALESIN, D. H., COOK, R. L. 1987. Rendering Antialiased Shadows with Depth Maps. Computer Graphics (SIGGRAPH '87 Proceedings), pp. 238-291.

RIGUER, G., TATARCHUK, N., ISIDORO, J. 2003. Real-Time Depth of Field Simulation. ShaderX^2: Shader Programming Tips and Tricks With DirectX 9.0. W., Ed. Wordware

SCHEUERMANN, T., TATARCHUK, N. 2004. Improved Depth of Field Rendering. ShaderX^3: Advanced Rendering with DirectX and OpenGL. Engel, W., Ed. Charles River Media

STARIK, S., AND WERMAN, M., 2003. Simulation of Rain in Videos. Texture 2003 (The 3rd international workshop on texture analysis and synthesis)

TOYSHOP DEMO, 2005. ATI Research, Inc. Can be downloaded from http://www.ati.com/developer/demos/rx1800.html

VAN DE HULST, H. C. 1981. Light Scattering by Small Particles. Dover Publications.

WANG, T. AND CLIFFORD, R. S. 1975. Use of Rainfall-Induced Optical Scintillations to Measure Path-Averaged Rain Parameters. JOSA, pp. 8-927-237.

Chapter 4

Rendering Gooey Materials with Multiple Layers

Chris Oat[6]
ATI Research

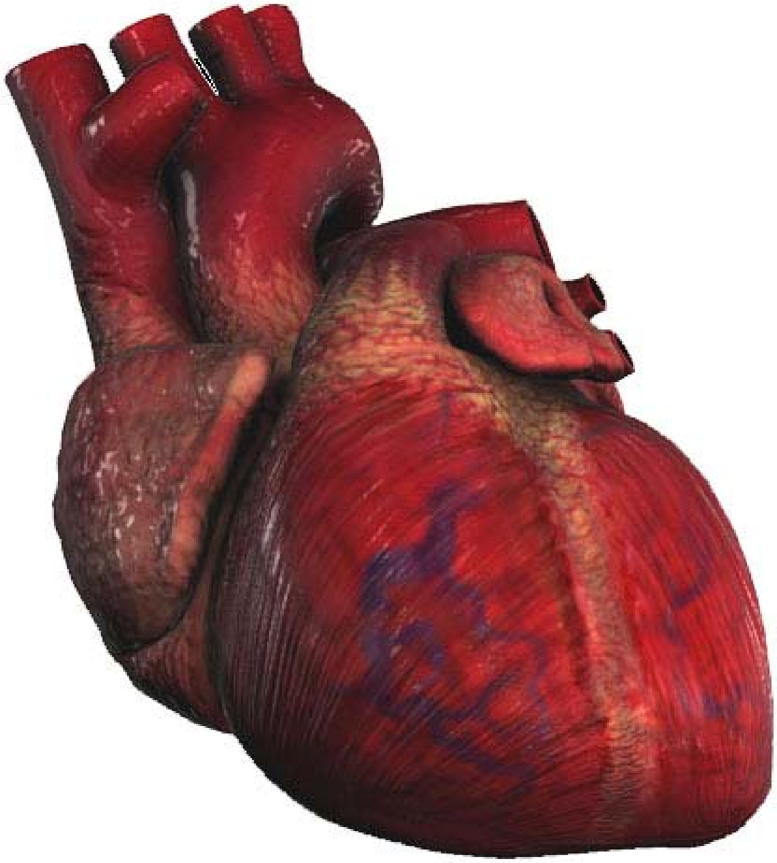


Figure 1. *A human heart rendered in real-time using the multi-layered shading technique described in this chapter.*

[6] coat@ati.com

4.1 Abstract

An efficient method for rendering semi-transparent, multi-layered materials is presented. This method achieves the look of a volumetric material by exploiting several perceptual cues, based on depth and illumination, while combining multiple material layers on the surface of an otherwise non-volumetric, multi-textured surface such as the human heart shown in Figure 1. Multiple implementation strategies are suggested that allow for different trade-offs to be made between visual quality and runtime performance.

4.2 Introduction

Creating immersive and compelling virtual environments requires a keen attention to detail. Over the years, much has been written about achieving photorealism in computer graphics and there is a plethora of information available for those interested in achieving realism through the use of physically based rendering techniques, but physical “correctness” comes at a price and frequently these techniques are difficult to apply or are too slow to evaluate in applications that demand interactive rendering. One area of computer graphics that frequently relies on computationally intensive, physically based rendering is that of volumetric material rendering.

Many real world surfaces, particularly biological materials, exhibit volumetric material properties due to their multiple, semi-transparent layers. Rendering these materials in a physically correct way can be very computationally demanding as it requires simulating complex light interactions due to the tendency of these materials to exhibit subsurface scattering. These physically based techniques are typically too expensive (both in terms of implementation time and render time) for use in applications such as games where “physical correctness” is far less important than rendering efficiency; in games it doesn’t matter how nice an image looks if it isn’t rendered fast enough to be interactive. We seek to approximate the overall look of volumetric, multi-layered and semi-transparent materials using efficient real-time rendering techniques that can easily be employed by interactive applications.

Instead of finding physically correct ways to simulate the properties of volumetric, multi-layered materials, we instead focus on recreating the important visual aspects of these kinds of materials:

- Inter-layer occlusion (opaque outer layers occlude inner layers)
- Depth parallax (parallax due to layer depth and thickness)
- Light diffusion (light scatters between layers)

We can achieve these properties by combining inexpensive rendering techniques such as: alpha compositing, parallax offsetting, normal mapping and image filtering.

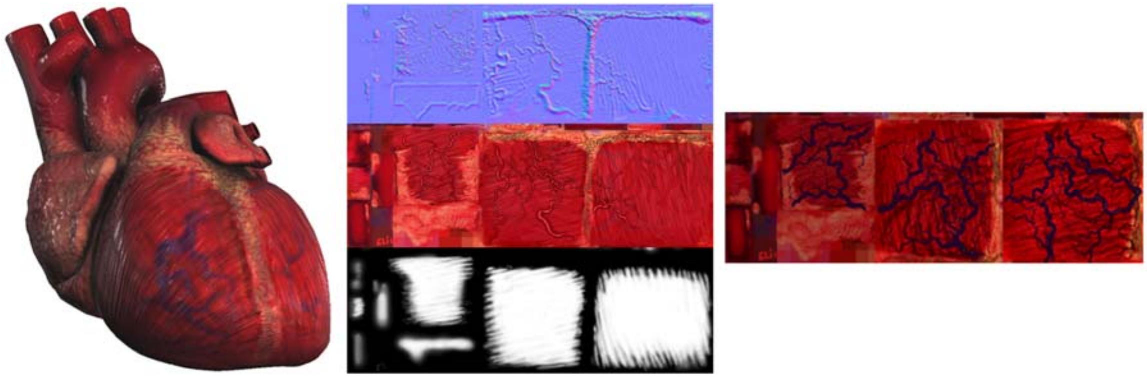


Figure 2. *[Left] Human heart rendered using a two layer material. [Middle] The material's outer layer is composed of a tangent space normal map (top), albedo map (center) and opacity map (bottom). [Right] The material's inner layer uses a single texture to store the inner layer's albedo.*

4.3 Multiple Layers

In computer graphics, volumetric materials are generally approximated with multiple, discrete layers (and often stored using 3D textures). The layers can be semi-transparent and each layer exhibits some amount of light reflectance, transmission, and absorption. We'll be using a two layer material to demonstrate this technique as it is applied to a model of a human heart but in general there's no limit to the number of material layers that may be used.

Our material's two layers are built up from the 2D texture maps shown in Figure 2. The material's outer surface layer is composed of an albedo map, a tangent space normal map for storing the surface detail, and a transparency map that stores the outer layer's varying opacity (more opaque in muscular regions and more transparent in the membrane/gelatinous regions). The material's inner layer requires only a single 2D texture for storing albedo; this layer stores the heart's subsurface vein networks.

In order to get the correct inter-layer occlusion, the outer layer's opacity value is used to blend the two layers together. Layer blending is achieved by performing a simple lerp (linear interpolation) between the two layers based on the outer layer's opacity map. Unfortunately, this simple composite doesn't give us a very volumetric looking material. The result is flat and unrealistic because it doesn't take the material's depth/thickness into account. One inexpensive way to create the impression of layer depth is to use a form of parallax offsetting.

4.4 Depth Parallax

In order to convince the viewer that he or she is looking into a volumetric material, it is important to capture the effects of parallax. Parallax, which is an important perceptual cue for determining distance and depth, is the apparent shift in an object's position relative to other objects due to changes in viewing position [Kaneko01] [Welsh04]. If we use the same texture coordinate to sample and composite the outer and inner layers' textures then the material won't appear volumetric. In order to create the illusion of depth, parallax offsetting is used to adjust the inner layer’s texture coordinate so that the inner layer’s texture is sampled in a perspective correct way. Figure 3 illustrates how our texture coordinate for sampling from the inner layer’s texture must be offset based on both the viewing angle and the distance between the layers.

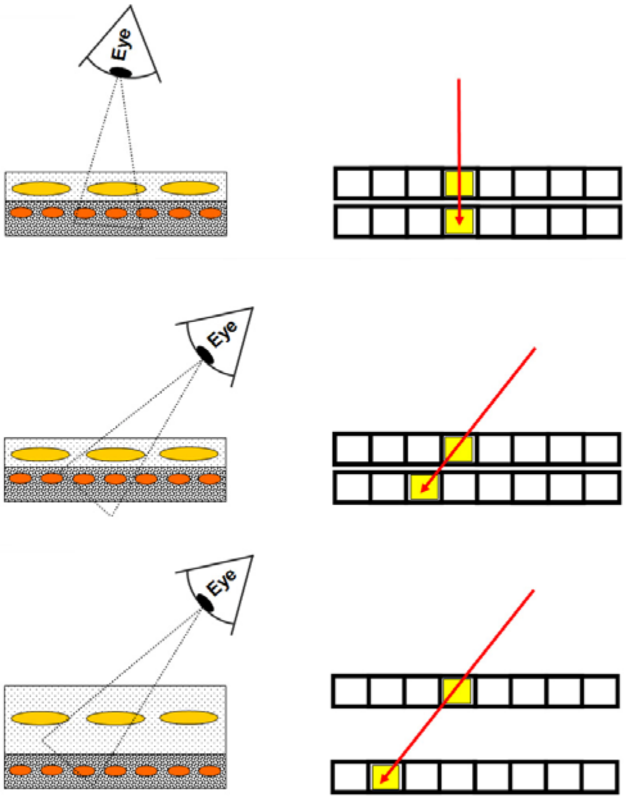
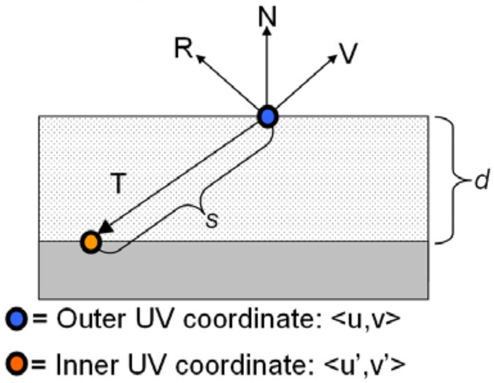


Figure 3. *[Top and Middle] Two different viewpoints of the same material. As the view position changes the sampling location of the inner layer shifts. [Bottom] The shift due to parallax is more pronounced as the thickness of the outer material layer increases.*

A new texture coordinate can be easily computed for the inner layer by making the simplifying assumption that the layers are parallel in tangent space, that is, the distance between the material layers is homogenous. This assumption allows us to avoid performing any kind of ray-tracing in order to find the correct surface point from which to sample the inner layer’s texture.



$$\vec{R} = -\vec{V} - 2 * dot(-\vec{V}, \vec{N}) * \vec{N}$$
$$\vec{T} = \langle \vec{R}_x, \vec{R}_y, -\vec{R}_z \rangle$$
$$s = d / \left|\vec{T}_z\right|$$
$$\langle u', v' \rangle = \langle u, v \rangle + s \langle \vec{T}_x, \vec{T}_y \rangle$$

Figure 4. *A new texture coordinate is computed for sampling from the inner layer's texture. The inner layer's texture coordinate is computed based on the viewing vector, outer layer's surface normal, and the layer thickness. The distance* ***d*** *between the outer layer and the inner layer is assumed to be homogenous for the entire material and all the vectors are assumed to be unit-length and in tangent space.*

In order to calculate the inner layer's texture coordinate, we must first compute a transmission vector. The transmission vector points from the outer surface layer's sample point to the inner layer's sample point (as shown in Figure 4). We start by computing a reflection vector, which is the view vector reflected about the per-pixel surface normal (sampled from the outer layer's normal map). Since we're working in tangent space, the transmission vector is simply the reflection vector with its z component negated such that it points down into the material. The transmission distance ***s*** (that's the distance along the unit-length transmission vector from the outer layer to the inner layer) is then computed by dividing the layer thickness ***d*** by the z component of the unit-length transmission vector. It's important to note that the transmission distance and layer thickness values are in texture space and therefore are expressed in "texel units" (a texel's width and height are 1/texture width and 1/texture height).

Once we know the transmission vector and the transmission distance, we can find the inner layer's offset texture coordinate by computing the intersection of the transmission vector with the inner surface layer. This new texture coordinate is then used for sampling the inner layer's texture in a perspective correct way, as shown in Figure 5.

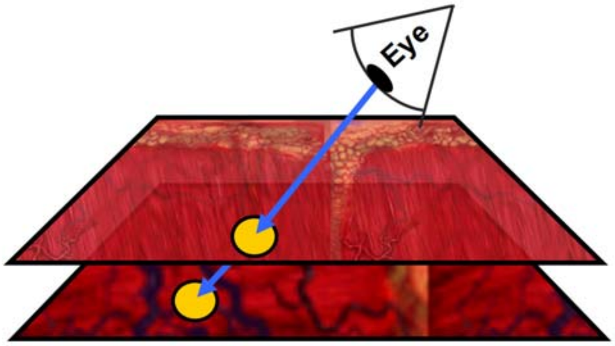


Figure 5. *The parallax offset texture coordinate is used for sampling the inner layer's albedo texture.*

Using a parallax offset texture coordinate to sample the inner layer’s albedo texture will provide a fairly convincing sense of material depth and thickness. An HLSL implementation of the parallax offsetting function is provided in listing 1. This function takes a number of parameters and returns a float3 vector that contains the new texture coordinate for sampling from the inner layer’s texture as well as the transmission distance through the material. The transmission distance will be used later on for computing lighting.

```
// Compute inner layer’s texture coordinate and transmission depth
// vTexCoord: Outer layer’s texture coordinate
// vViewTS: View vector in tangent space
// vNormalTS: Normal in tangent space (sampled normal map)
// fLayerThickness: Distance from outer layer to inner layer
float3 ParallaxOffsetAndDepth ( float2 vTexCoord, float3 vViewTS,
                                float3 vNormalTS, float fLayerThickness )
{
       // Tangent space reflection vector
       float3 vReflectionTS = reflect( -vViewTS, vNormalTS );

       // Tangent space transmission vector (reflect about surface plane)
       float3 vTransTS = float3( vReflectionTS.xy, -vReflectionTS.z );

       // Distance along transmission vector to intersect inner layer
       float fTransDist = fLayerThickness / abs(vTransTS.z);

       // Texel size: Hard coded for 1024x1024 texture size
       float2 vTexelSize = float2( 1.0/1024.0, 1.0/1024.0 );

       // Inner layer’s texture coordinate due to parallax
       float2 vOffset = vTexelSize * fTransDist * vTransTS.xy;
       float2 vOffsetTexCoord = vTexCoord + vOffset;

       // Return offset texture coordinate in xy and transmission dist in z
       return float3( vOffsetTexCoord, fTransDist );
}
```

Listing 1. *An example HLSL implementation of the Parallax offsetting code. Given a texture coordinate on the outer surface layer, a tangent space view and normal vectors, and a layer thickness parameter, this function returns the transmission distance and offset texture coordinates for sampling from the inner layer’s texture map.*

4.5 Light Scattering

In addition to parallax and perspective, a material’s reflectance properties provide important visual cues about the material too. In order to achieve a convincing volumetric look, it is important to approximate the effects of light scattering as it passes through our volumetric, multi-layered material. Once again, instead of focusing on physically correct methods – for light transport through scattering media – we simply focus our efforts on recreating the observable results of this complex behavior.

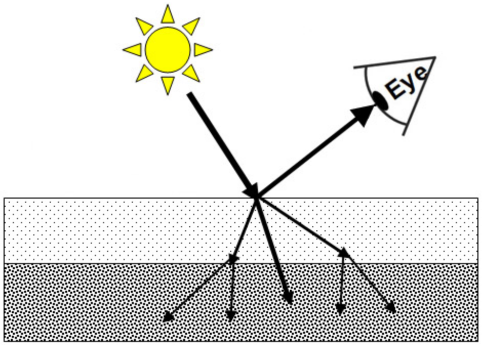


Figure 6. *Light reaches the surface of a material. Some of the light is reflected directly back to the viewer and some of the light is scattered into the material. Since the light that is scattered into the material arrives at inner material layers from many different (scattered) directions, the inner layer will appear more evenly lit (almost as if it were lit by many small light sources from many different directions)*

Figure 6 provides a simplified view of a material that scatters incoming light. In this diagram, some of the light that reaches the material's surface is directly reflected back to the viewer and the rest of the light is scattered down into the material to its subsurface layer.

The outer surface layer's illumination can be approximated using a standard local diffuse lighting model (using a surface normal sampled from the outer layer's normal map to compute N·L). This local lighting term accounts for the light that reaches the surface and is reflected back to the viewer directly (i.e. it doesn't participate in any subsurface scattering). The light that isn't directly reflected back to the viewer is transmitted into the material. This transmitted light may be scattered multiple times before it reaches the material's inner layer and thus will arrive at the inner layer from many different directions (not necessarily the same direction from which it originally arrived at the surface). The over-all effect of this scattering is that the material's inner layer appears more evenly lit – as if it were lit by many small light sources – since it's lit from many different directions. We can achieve this low-frequency lighting effect by rendering the surface layer's diffuse lighting to an off-screen render target and applying a blur kernel to filter out the high-frequency lighting details.

Texture space lighting techniques have been shown to effectively simulate the effects of subsurface scattering in human skin for both offline and real-time applications [Borshukov03] [Sander04]. We use this same basic technique to simulate light scattering as it enters our multi-layered material. First we render our mesh into an off-screen buffer using the mesh's texture coordinates as projected vertex positions (this allows the rasterizer to un-wrap our mesh into texture space as shown in Figure 7).

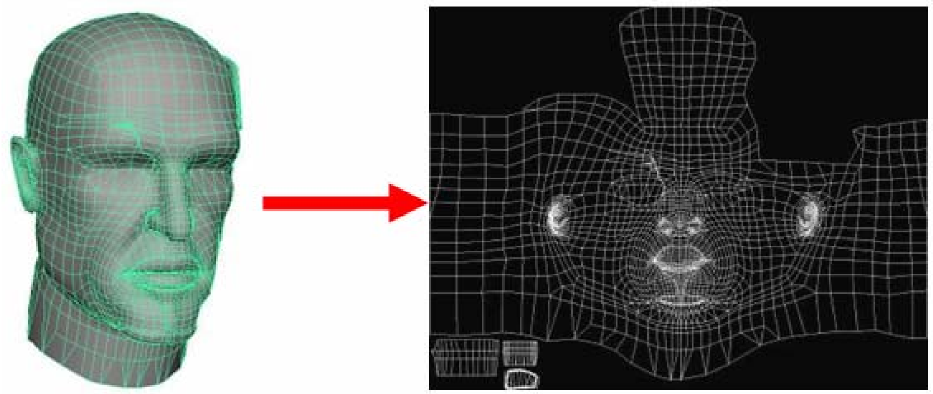


Figure 7. *A mesh is rendered into an off-screen buffer using its texture coordinates as position [Gosselin04].*

The mesh is shaded using a local diffuse lighting model using its 3D world space positions but is rendered into texture space by scaling its texture coordinates such that they're in the correct NDC screen space. The resulting image can be thought of as a dynamic light map for the outer layer of our multi-layered material. We can then use a blurred version of this light map for our material's inner layer (see Figure 8). This gives us smooth, diffused lighting on the material's inner layer. The degree to which the light map needs to be blurred depends on the thickness and scattering properties of the outer layer (the more the outer layer scatters light, the more we need to blur the light map).

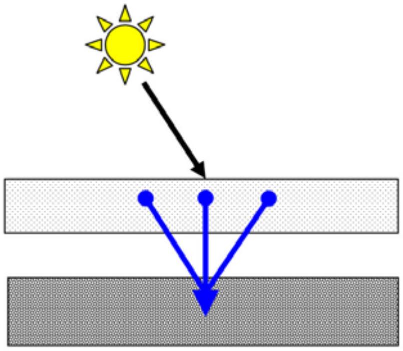


Figure 8. *The average of multiple samples taken from the outer layer's light map is used to light the inner layer. By blurring the outer layer's light map and applying it to the inner layer, we get a nice low-frequency lighting term for the inner layer which creates the appearance of subsurface scattering.*

A resizable Poisson disc filter kernel is ideal for performing the light map blurring since it can be resized dynamically based on the amount of light scattering we wish to simulate. This blur kernel takes a fixed number of taps, where the taps are distributed randomly on a unit disc following a Poisson distribution, from the source texture (see Figure 9). The kernel's size (or the disc's radius, if you prefer) can be scaled on a per-pixel basis to provide more or less blurring as needed. Our kernel's size is proportional to the outer layer's thickness; a thicker outer layer will result in a larger kernel and thus more blurring/scattering.

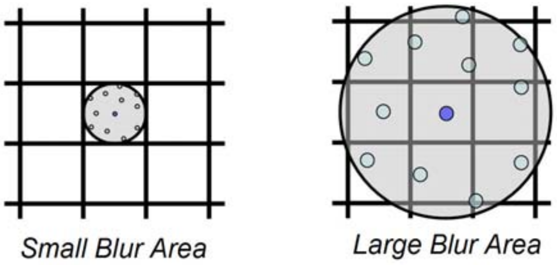


Figure 9. *A resizable Poisson disc filter kernel is useful in applications where the amount of image blurring is dynamic. The individual tap locations are fixed with respect to the disc but the disc itself can be resized to achieve varying amounts of image blurring.*

This blur kernel is straightforward to implement as an HLSL function (listing 2). The individual tap locations are pre-computed for a unit disc and are hard-coded in the function. This function can then be used to compute a dynamic light map for our material's inner layer.

```
// Growable Poisson disc (13 samples)
// tSource: Source texture sampler
// vTexCoord: Texture space location of disc’s center
// fRadius: Radius if kernel (in texel units)
float3 PoissonFilter ( sampler tSource, float2 vTexCoord, float fRadius )
{
   // Hard-coded texel size: Assumes 1024x1024 source texture
   float2 vTexelSize = float2( 1.0/1024.0, 1.0/1024.0 );

   // Tap locations on unit disc
   float2 vTaps[12] = {float2(-0.326212,-0.40581),float2(-0.840144,-0.07358),
                       float2(-0.695914,0.457137),float2(-0.203345,0.620716),
                       float2(0.96234,-0.194983), float2(0.473434,-0.480026),
                       float2(0.519456,0.767022), float2(0.185461,-0.893124),
                       float2(0.507431,0.064425), float2(0.89642,0.412458),
                       float2(-0.32194,-0.932615),float2(-0.791559,-0.59771)};

   // Take a sample at the disc’s center
   float3 cSampleAccum = tex2D( tSource, vTexCoord );

   // Take 12 samples in disc
   for ( int nTapIndex = 0; nTapIndex < 12; nTapIndex++ )
   {
      // Compute new texture coord inside disc
      float2 vTapCoord = vTexCoord + vTexelSize * vTaps[nTapIndex] * fRadius;

      // Accumulate samples
      cSampleAccum += tex2D( tSource, vTapCoord );
   }

   return cSampleAccum / 13.0; // Return average
}
```

Listing 2. *An HLSL function that implements a resizable Poisson disc filter kernel. The source texture's texel size has been hard-coded but could also be passed as an additional argument to the function.*

The filtered light map approximates light scattering as it enters the volumetric material but we must also account for light scattering on its way back out of the material (Figure 10).

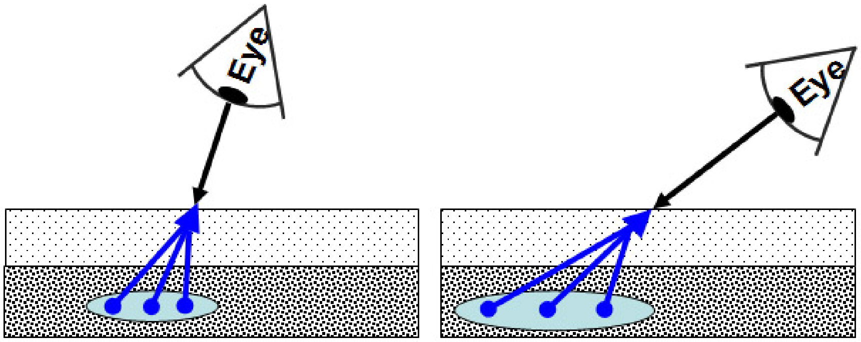


Figure 10. *The Poisson disc sampling function is used for sampling the inner layer's albedo texture to approximate light that is reflected back to the surface, possibly scattering multiple times before it re-emerges at the point where the viewer is looking. The kernel is centered about the parallax offset texture coordinate and its size is a function of the amount of material the viewer is looking through (transmission distance).*

As light scatters down into subsurface material layers, some of the light is reflected back by the material's inner layer. Just as that light scattered as it entered the material, the light also scatters on its way back out. The net result of this additional scattering is that the material's inner layer appears blurry. The deeper the inner layer is from the surface and the more edge-on the viewing angle is, the more material you have to look through to see the inner layer and thus the more blurry it should appear (it's had to scatter through more "stuff" to make it back to your eye). In order to achieve this effect, we use our Poisson disc filtering function again but this time we use it to sample from the inner layer's albedo map. The kernel should be centered about the parallax offset texture coordinate to get a perspective correct sampling location and the kernel's radius should be scaled depending on the transmission distance through the material (this distance is also computed by the parallax offset function from listing 1).

4.6 Putting It All Together

Using the above techniques, we have all the tools necessary to construct a volumetric looking multi-layered material. Our final task is to combine the various techniques into a single shader that can be applied to our mesh. Listing 3 provides HLSL code that uses all the techniques described above to construct a multi-layered material shader that was used to render the beating heart shown in Figure 11.

```
// Sample from outer layer’s base map and light map textures
float3 cOuterDiffuse = tex2D(tLightMap, i.vTexCoord);
float4 cOuterBase = tex2D(tOuterBaseMap, i.vTexCoord); // Opacity in alpha


// Compute parallax offset texture coordinate for sampling from
// inner layer textures, returns UV coord in X and Y and transmission
// distance in Z
float3 vOffsetAndDepth = ParallaxOffsetAndDepth(i.vTexCoord, vViewTS,
                                                vNormalTS, fLayerThicknes);


// Poisson disc filtering: blurry light map (blur size based on layer
// thickness)
float3 cInnerDiffuse = PoissonFilter( tLightMap,
                                      vOffsetAndDepth.xy,
                                      fLayerThickness );


// Poisson disc filtering: blurry base map (blur size based on
// transmission distance)
float3 cInnerBase = PoissonFilter( tInnerBaseMap,
                                   vOffsetAndDepth.xy,
                                   vOffsetAndDepth.z );


// Compute N.V for additional compositing factor (prefer outer layer
// at grazing angles)
float fNdotV = saturate( dot(vNormalTS, vViewTS) );


// Lerp based on opacity and N.V (N.V prevents artifacts when view
// is very edge on)
float3 cOut = lerp( cOuterBase.rgb * cOuterDiffuse.rgb,
                    cInnerBase.rgb * cInnerDiffuse.rgb,
                    cOuterBase.a * fNdotV );
```

Listing 3: *Parallax offsetting and Poisson disc image filtering are used to implement the final multi-layered material shader. This code snippet assumes that the texture-space lighting was performed in a previous pass, the results of which are available via the tLightMap texture sampler.*

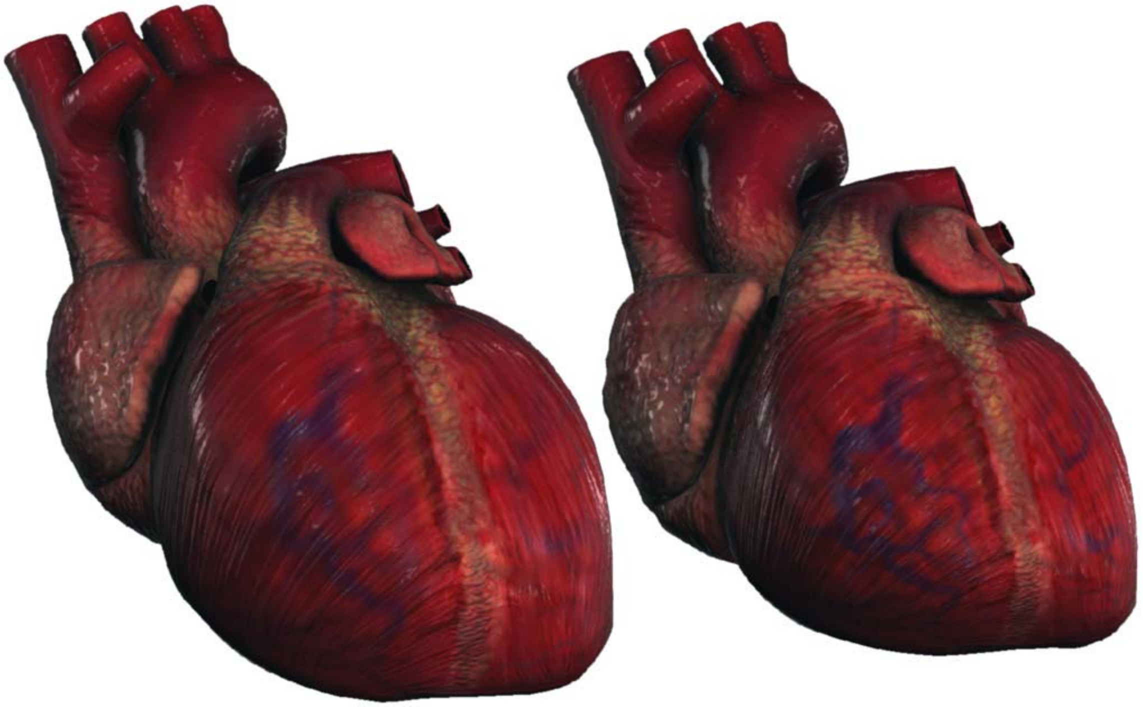


Figure 11. *Two frames of a beating human heart animation that was rendered using the techniques described in this chapter. As the heart expands (left) the distance between material layers is increased which results in a higher degree of depth parallax between layers and a larger blur kernel is used for sampling the dynamic light map and the inner layer's albedo map. When the heart is contracted (right) the layer depth is decreased and the inner material layer is more discernable.*

4.6 Optimizations

The multi-layered material shading technique that's been presented in this chapter has several properties that may make it prohibitively expensive for use in certain rendering scenarios. Because the technique uses a texture-space lighting approach for subsurface scattering, rendering with this technique requires rendering to an off-screen texture which implies making render target changes. Each object in a scene that uses this effect would require its own off-screen render target and thus make batched rendering very difficult if not impossible (this is really only a problem if you plan to apply this effect to many different objects that will be rendered on screen at the same time). In addition to the render target switching overhead, this technique requires significant texturing bandwidth for the multiple texture samples that must be taken from both the outer layer's dynamic light map and the inner layer's albedo map. Two optimizations are suggested that eliminate one or both of these costs by sacrificing some image quality for increased rendering performance.

Our first optimization eliminates the need for a dynamic light map and thus eliminates the need to create and render to an off-screen buffer. The main goal of using the texture space lighting technique to generate a dynamic light map is that it may be used for computing a low-frequency lighting term that may be applied to the material's inner layer.

The dynamic light map essentially allows us to compute lighting for the outer layer once and then re-use these results multiple times when computing the inner layer's lighting term. But we can also achieve a similar effect by taking multiple samples from the outer layer's normal map and computing a bunch of lighting terms on the fly that are then averaged and applied to the inner layer (Figure 12). A new Poisson disc filtering function could be written for sampling from the outer layer's normal map, this new function would compute a diffuse lighting term (N·L) for each tap taken from the normal map and would then average the resulting diffuse lighting terms (we aren't averaging the normals themselves). As before, the filter kernel could be resized depending on the thickness of the various material layers. This optimization entirely eliminates the need for an off-screen renderable texture but still delivers the desired result: high-frequency lighting detail on the outer layer and low-frequency (blurry) lighting on the inner layer.

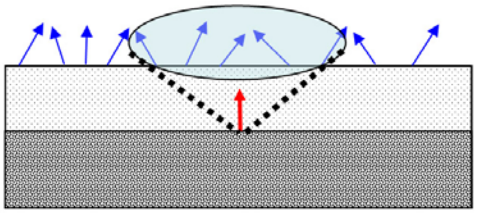


Figure 12. *Multiple samples are taken from the outer layer's normal map (blue vectors) and are used to compute multiple diffuse lighting terms. The resulting diffuse lighting terms are then averaged and used as a final diffuse lighting term for the inner layer. This gives the desired result of a high-frequency outer-layer lighting term and a low-frequency inner-layer lighting term.*

This optimization is not perfect though, it results in a lot of redundant math since diffuse lighting gets computed multiple times for a given normal in the normal map and it does nothing for reducing the amount of texture sampling since we have effectively traded light map sampling for normal map sampling.

In order to make further performance improvements we could eliminate much of the required texture look ups by taking advantage of the mesh's per-vertex normals. Instead of using a filter kernel to achieve low-frequency lighting on the material's inner layer, we instead use the mesh's interpolated vertex normals for computing the inner layer's diffuse lighting term (Figure 13). This drastically reduces the amount of texture bandwidth needed by our shader and gives us more-or-less what we're aiming for; high-frequency lighting detail on the outer layer and smoother, lower-frequency lighting on the inner layer.

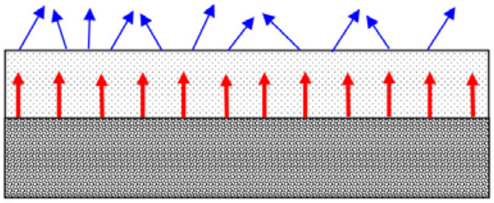


Figure 13. *The outer layer's diffuse lighting is computed using a per-pixel normal (blue) sampled from a normal map for high-frequency lighting detail. The inner layer's diffuse lighting is computed using a per-vertex normal for coarser, lower-frequency lighting.*

4.7 Future Directions

The multi-layered material shader presented in this chapter provides the basic look and feel of a multi-layered, volumetric material but there are many ways in which it might be improved to make it appear more realistic. The example material we've used makes use of two material layers to shade a model of a human heart, but the technique could be applied to materials with many layers by making use of the same kinds of tricks and techniques. Also, the subject of inter-layer shadowing from opaque material layers has been completely ignored. Inter-layer shadowing could potentially be implemented by adjusting the samples taken from the dynamic light map based on the outer layer's opacity. This would keep light from passing through opaque regions of the outer layer and onto inner layer regions that should appear to be in shadow. The shadowing probably becomes more important when you're dealing with many material layers that are very deep/thick since in that scenario it would be possible and even likely that a viewer, looking edge-on into the material, would see below opaque outer regions where one would expect to see inter-layer shadowing occur.

4.8 Conclusion

A method for rendering volumetric, multi-layered materials at interactive frame rates has been presented. Rather than focus on physically correct simulations, this technique exploits various perceptual cues based on viewing perspective and light scattering to create the illusion of volume when combining multiple texture layers to create a multi-layered material. In an attempt to make the technique scalable across a number of different platforms with various rendering performance strengths and weaknesses, two optimizations have been provided that allow an implementation to trade visual quality for performance where appropriate.

4.9 References

BORSHUKOV, G., AND LEWIS, J. P. 2003. Realistic Human Face Rendering for The Matrix Reloaded. In the proceedings of SIGGRAPH 2003, Technical Sketches

GOSSELIN, D. 2004. Real Time Skin Rendering. Game Developer Conference, D3D Tutorial 9

KANEKO, T., TAKAHEI, T., INAMI, M., KAWAKAMI, N., YANAGIDA, Y., MAEDA, T., TACHI, S. 2001. Detailed Shape Representation with Parallax Mapping. In Proceedings of ICAT 2001, pp. 205-208.

SANDER, P.V., GOSSELIN, D., AND MITCHELL, J. L.2004. Real-Time Skin Rendering on Graphics Hardware. In the proceedings of SIGGRAPH 2004, Technical Sketch.

WELSH, T. 2004. Parallax Mapping, ShaderX3: Advanced Rendering with DirectX and OpenGL, Engel, W. Ed., A.K. Peters, 2004

Chapter 5

Practical Parallax Occlusion Mapping with Approximate Soft Shadows for Detailed Surface Rendering

Natalya Tatarchuk[7]
ATI Research

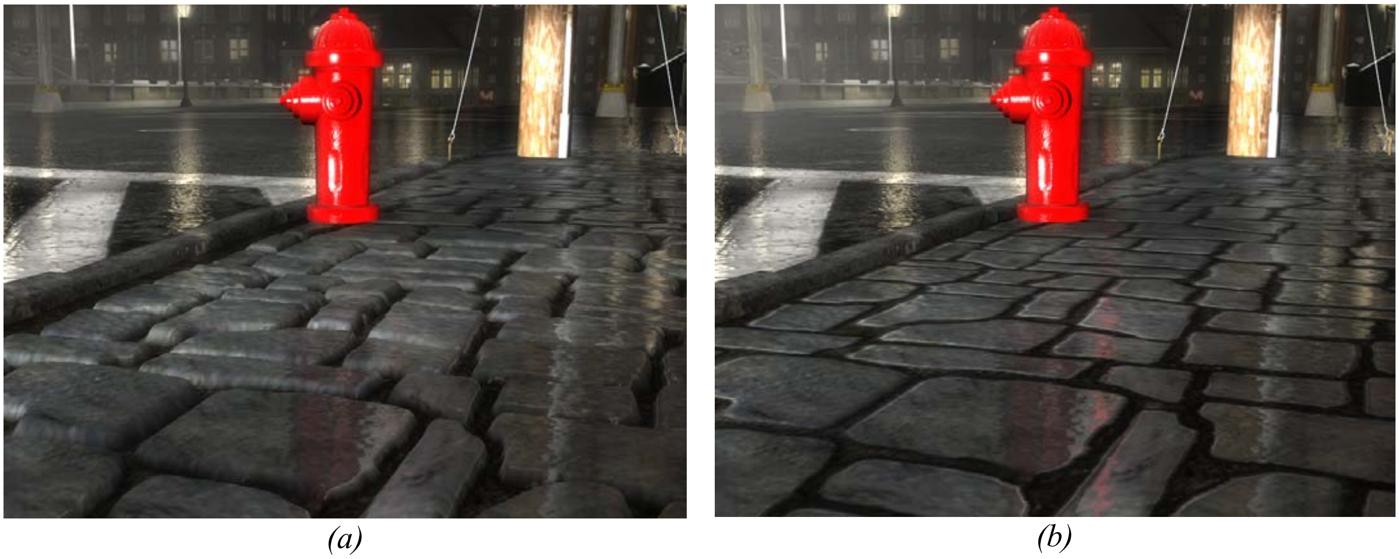


***Figure 1**. Realistic city scene rendered using parallax occlusion mapping applied to the cobblestone sidewalk in (a) and using the normal mapping technique in (b).*

[7] natasha@ati.com

5.1 Abstract

This chapter presents a per-pixel ray tracing algorithm with dynamic lighting of surfaces in real-time on the GPU. First, we will describe a method for increased precision of the critical ray-height field intersection and adaptive height field sampling. We achieve higher quality results than the existing inverse displacement mapping algorithms. Second, soft shadows are computed by estimating light visibility for the displaced surfaces. Third, we describe an adaptive level-of-detail system which uses the information supplied by the graphics hardware during rendering to automatically manage shader complexity. This LOD scheme maintains smooth transitions between the full displacement computation and a simplified representation at a lower level of detail without visual artifacts. Finally, algorithm limitations will be discussed along with the practical considerations for integration into game pipelines. Specific attention will be given to the art asset authoring, providing guidelines, tips and concerns. The algorithm performs well for animated objects and supports dynamic rendering of height fields for a variety of interesting displacement effects. The presented method is scalable for a range of consumer grade GPU products. It exhibits a low memory footprint and can be easily integrated into existing art pipelines for games and effects rendering.

5.2 Introduction

The advances in the programmability of commodity GPUs in the recent years have revolutionized the visual complexity of interactive worlds found in games or similar real-time applications. However, the balance between the concept and realism dictates that in order to make the objects in these virtual worlds appear photorealistic, the visual intricacy demands a significant amount of detail. Simply painting a few broken bricks will not serve the purpose of displaying a dilapidated brick wall in a forlorn city any longer. With the raised visual fidelity of the latest games, the player wants to be immersed in these worlds – they want to *experience* the details of their environments. That demands that each object maintains its three-dimensional appearance accurately regardless of the viewing distance or angle.

Which brings us to an age-old problem of computer graphics - how do we render detailed objects with complex surface detail without paying the price on performance? We must balance the desire to render intricate surfaces with the cost of the millions of triangles associated with high polygonal surfaces typically necessary to represent that geometry. Despite the fact that the geometric throughput of the graphics hardware has increased immensely in recent years, there still exist many obstacles in throwing giant amounts of geometry onto the GPU. There is an associated memory footprint for storing large meshes (typically measured in many megabytes of vertex and connectivity data), and the performance cost for vertex transformations and animations of those meshes.

If we want the players to think they're near a brick wall, it should look and behave like one. The bricks should have deep grooves, an assortment of bumps and scratches. There should be shadows between individual bricks. As the player moves around the object, it needs to maintain its depth and volume. We wish to render these complex surfaces, such as this mythical brick wall, accurately – which means that we must do the following:

- Preserve depth at all angles
- Support dynamic lighting
- Self occlusions on the surface must result in correct self-shadowing on the surface without aliasing.

Throughout history, artists specialized in creating the illusion of detail and depth without actually building a concrete model of reality on the canvas. Similarly, in computer graphics we frequently want to create a compelling impression of a realistic scene without the full cost of complex geometry. Texture mapping is essential for that purpose - it allows generation of detail-rich scenes without the full geometric content representation. Bump mapping was introduced in the early days of computer graphics in [Blinn 1978] to avoid rendering high polygonal count models.

Bump mapping is a technique for making surfaces appear detailed and uneven by perturbing the surface normal using a texture. This approach creates a visual illusion of the surface detail that would otherwise consume most of a project's polygon budget (such as fissures and cracks in terrain and rocks, textured bark on trees, clothes, wrinkles, etc). Since the early days, there have been many extensions to the basic bump mapping technique including emboss bump mapping, environment map bump mapping, and the highly popular dot product bump mapping (normal mapping). See [Akenine-Möller02] for a more detailed description of these techniques. In Figure 1b above, we can see the per-pixel bump mapping technique (also called *normal mapping*) applied to the cobblestone sidewalk.

Despite its low computational cost and ease of use, bump mapping fails to account for important visual cues such as shading due to interpenetrations and self-occlusion, nor does it display perspective-correct depth at all angles. Since the bump mapping technique doesn't take into consideration the geometric depth of the surface, it does not exhibit parallax. This technique displays various visual artifacts, and thus several approaches have been introduced to simulate parallax on bump mapped geometry. However, many of the existing parallax generation techniques cannot account for self-occluding geometry or add shadowing effects. Indeed, shadows provide a very important visual cue for surface detail.

The main contribution of this chapter is an advanced technique for simulating the illusion of depth on uneven surfaces without increasing the geometric complexity of rendered objects. This is accomplished by computing a perspective-correct representation maintaining accurate parallax by using an inverse displacement mapping technique. We also describe a method for computing self-shadowing effects for self-occluding objects. The resulting approach allows us to simulate pseudo geometry displacement in the pixel shader instead of modeling geometric details in the polygonal mesh. This allows us to render surface detail providing a convincing visual impression of depth from varying viewpoints, utilizing the programmable pixel pipelines of commercial graphics hardware. The results of applying parallax occlusion mapping can be seen in Figure 1a above, where the method is used to render the cobblestone sidewalk.

We perform per-pixel ray tracing for inverse displacement mapping with an easy-to-implement, efficient algorithm. Our method allows interactive rendering of displaced surfaces with dynamic lighting, soft shadows, self-occlusions and motion parallax. Previous methods displayed strong aliasing at grazing angles, thus limiting potential applications' view angles, making these approaches impractical in realistic game scenarios. We present a significant improvement in the rendering quality necessary for production level results. This work has been originally presented in [Tatarchuk06].

Our method's contributions include:

- A high precision computation algorithm for critical height field-ray intersection and an adaptive height field sampling scheme, well-designed for a range of consumer GPUs (Section 6.5.1). This method significantly reduces visual artifacts at oblique angles.
- Estimation of light visibility for displaced surfaces allowing real-time computation of soft shadows due to self-occlusion (Section 6.5.2).
- Adaptive level-of-detail control system with smooth transitions (Section 6.5.3) for controlling shader complexity using per-pixel level-of-detail information.

The contributions presented in this chapter are desired for easy integration of inverse displacement mapping into interactive applications such as games. They improve resulting visual quality while taking full advantage of programmable GPU pixel and texture pipelines' efficiency. Our technique can be applied to animated objects and fits well within established art pipelines of games and effects rendering. The algorithm allows scalability for a range of existing GPU products.

5.3 Why reinvent the wheel? Common artifacts and related work

Although standard bump mapping offers a relatively inexpensive way to add surface detail, there are several downsides to this technique. Common bump mapping approaches lack the ability to represent view-dependent unevenness of detailed surfaces, and therefore fail to represent motion parallax—the apparent displacement of the object due to viewpoint change. In recent years, new approaches for simulating displacement on surfaces have been introduced. [Kaneko01] and [Welsh03] describe an approach for parallax mapping for representing surface detail using normal maps, while [Wang03] introduced a technique for view-dependent displacement mapping which improved on displaying surface detail as well as silhouette detail.

Displacement mapping, introduced by [Cook84], addressed the issues above by actually modifying the underlying surface geometry. Ray-tracing based approaches dominated in the offline domain [Pharr and Hanrahan 1996; Heidrich and Seidel 1998]. These methods adapt poorly to current programmable GPUs and are not applicable to the interactive domain due to high computational costs. Additionally, displacement mapping requires fairly highly tessellated models in order achieve satisfactory results, negating the polygon-saving effect of bump mapping.

Other approaches included software-based image-warping techniques for rendering perspective-correct geometry [Oliveira et al. 2000] and precomputed visibility information [Wang et al. 2003; Wang et al. 2004; Donnelly 2005]. [Wang03] describes a per-pixel technique for self-shadowing view-dependent rendering capable of handling occlusions and correct display of silhouette detail. The precomputed surface description is stored in multiple texture maps (the data is precomputed from a supplied height map). The view-dependent displacement mapping textures approach displays convincing parallax effect by storing the texel relationship from several viewing directions. However, the cost of storing multiple additional texture maps for surface description is prohibitive for most real-time applications. Our proposed method requires a low memory footprint and can be used for dynamically rendered height fields.

Recent inverse displacement mapping approaches take advantage of the parallel nature of GPU pixel pipelines to render displacement directly on the GPU ([Doggett and Hirche 2000; Kautz and Seidel 2001; Hirche et al. 2004; Brawley and Tatarchuk 2004; Policarpo et al. 2005]. One of the significant disadvantages of these approaches is the lack of correct object silhouettes since these techniques do not modify the actual geometry. Accurate silhouettes can be generated by using view-dependent displacement data as in [Wang et al. 2003; Wang et al. 2004] or by encoding the surface curvature information with quadric surfaces as in [Oliveira and Policarpo 2005].

Another limitation of bump mapping techniques is the inability to properly model self-shadowing of the bump mapped surface, adding an unrealistic effect to the final look. The horizon mapping technique ([Max 1988], [Sloan and Cohen 2000]) allows shadowing bump mapped surfaces using precomputed visibility maps. With this approach, the height of the shadowing horizon at each point on the bump map for eight cardinal directions is encoded in a series of textures which are used to determine the amount of self-shadowing for a given light position during rendering. A variety of other techniques were introduced for this purpose, again, the reader may refer to an excellent survey in [Akenine-Möller02].

A precomputed three-dimensional distance map for a rendered object can be used for surface extrusion along a given view direction ([Donnelly 2005]). This technique stores a ‘slab’ of distances to the height field in a volumetric texture. It then uses this distance field texture to perform ray “walks” along the view ray to arrive at the displaced point on the extruded surface. The highly prohibitive cost of a 3D texture and dependent texture fetches’ latency make this algorithm less attractive and in many cases simply not applicable in most real-time applications. Additionally, as this approach does not compute an accurate intersection of the rays with the height field, it suffers from aliasing artifacts at a large range of viewing angles. Since the algorithm requires precomputed distance fields for each given height field, it is not amenable for dynamic height field rendering approaches.

Mapping relief data in tangent space for per-pixel displacement mapping in real-time was proposed in [Brawley and Tatarchuk 2004; Policarpo et al. 2005; McGuire and McGuire 2005] and further extended in [Oliveira and Policarpo et al. 2005] to support silhouette generation. The latter work was further extended to support rendering with non-height field data in [Policarpo06]. These methods take advantage of the programmable pixel pipeline efficiency by performing height field-ray intersection in the pixel shader to compute the displacement information. These approaches generate dynamic lighting with self-occlusion, shadows and motion parallax. Using the visibility horizon to compute hard shadows as in [Policarpo et al. 2005; McGuire and McGuire 2005; Oliveira and Policarpo

2005] can result in shadow aliasing artifacts. All of the above approaches exhibit strong aliasing and excessive flattening at steep viewing angles. No explicit level of detail schemes were provided with these approaches, relying on the texture filtering capabilities of the GPUs.

Adaptive level-of-detail control systems are beneficial any computationally intensive algorithm and there have been many contributors in the field of rendering. A level of detail system for bump mapped surfaces using pre-filtered maps was presented in [Fournier 92]. RenderMan® displacement maps were automatically converted to bump maps and BRDF representations in [Becker and Max 1993]. An automatic shader simplification system presented in [Olano et al. 2003] uses controllable parameters to manage system complexity. The resulting level-of-detail shader appearance is adjustable based on distance, size, and importance and given hardware limits.

5.4 Parallax Occlusion Mapping

This section will provide a brief overview of concepts of the parallax occlusion mapping method. We encode the displacement information for the surface in a height map as shown in Figure 2b. The inherent planarity of the tangent space allows us to compute displacement for arbitrary polygonal surfaces. Height field-ray intersections are performed in tangent space. The lighting can be computed in any space using a variety of illumination models. Efficient GPU implementation allows us to compute per-pixel shading, self-occlusion effects, and soft shadows, dynamically scaling the computations.

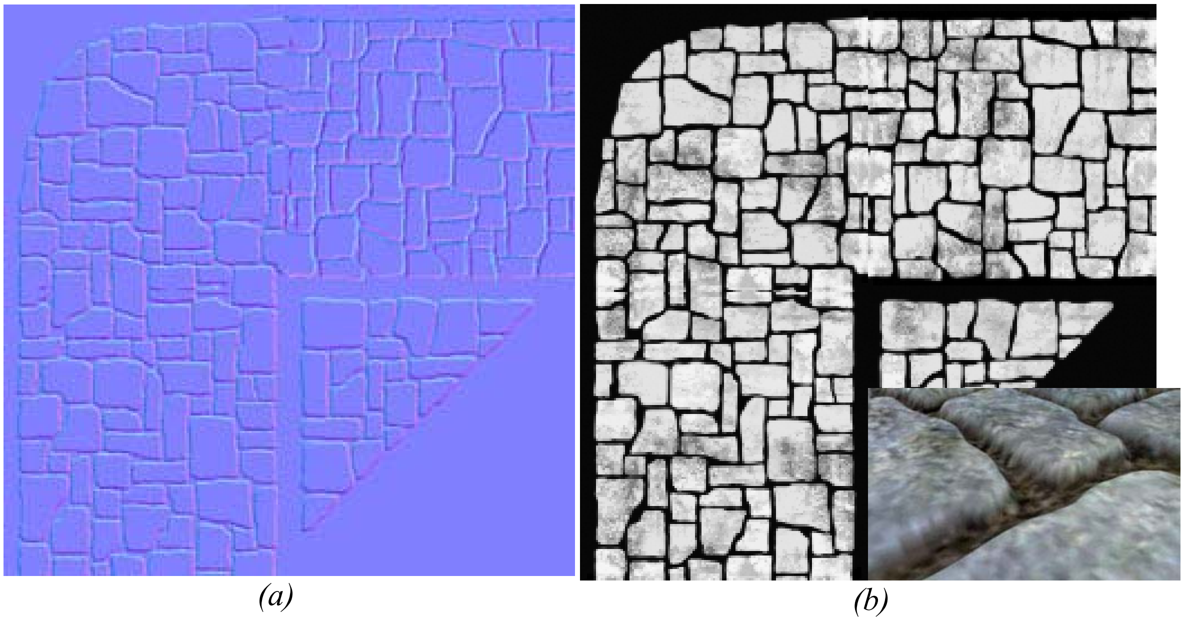


Figure 2. *(a) Tangent space normal map used to render the cobblestone sidewalk in Figure 1a. (b) Corresponding height field encoding the displacement information in the range [0;1] for that sidewalk object and a close-up view of the rendered sidewalk*

The effect of motion parallax for a surface can be computed by applying a height map and offsetting each pixel in the height map using the geometric normal and the view vector. As we move the geometry away from its original position using that ray, the parallax is obtained by the fact that the highest points on the height map would move the farthest along that ray and the lower extremes would not appear to be moving at all. To obtain satisfactory results for true perspective simulation, one would need to displace every pixel in the height map using the view ray and the geometric normal. We trace a ray through the height field to find the closest visible point on the surface.

The input mesh provides the reference plane for displacing the surface downwards. The height field is normalized for correct ray-height field intersection computation (0 representing the reference polygon surface values and 1 representing the extrusion valleys).

The parallax occlusion mapping algorithm execution can be summarized as follows:

- Compute the tangent-space viewing direction $\hat{V}_{ts}$ and the light direction $\hat{L}_{ts}$ per-vertex, interpolate and normalize in the pixel shader
- Compute the parallax offset vector P (either per-vertex or per-pixel) to determine the maximum visual offset in texture-space for the current level (as described in [Brawley and Tatarchuk 2004])

In the pixel shader:

- Ray cast the view ray $\hat{V}_{ts}$ along P to compute the height profile–ray intersection point. We sample the height field profile along P to determine the correct displaced point on the extruded surface. This yields the texture coordinate offset necessary to arrive at the desired point on the extruded surface as shown in Figure 3. We add this parallax offset amount to the original sample coordinates to yield the shifted texture coordinates t_{off}

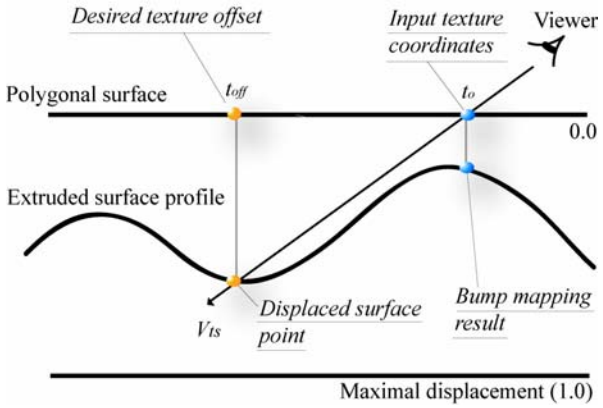


Figure 3. *Displacement based on sampled height field and current view direction.*

- Estimate the light visibility coefficient v by casting the light direction ray $\hat{L}_{ts}$ and sampling the height profile for occlusions.
- Shade the pixel using v, $\hat{L}_{ts}$ and the pixel's attributes (such as the albedo, color map, normal, etc.) sampled at the texture coordinate offset t_{off}.

Figure 6 later in the chapter illustrates the process above for a given pixel on a polygonal face. We will now discuss each of the above steps in greater detail.

5.4.1 Height Field – Ray Intersection

Techniques such as [Policarpo et al. 2005; Oliveira and Policarpo 2005] determine the intersection point by a combination of linear and binary search routines. These approaches sample the height field as a piecewise constant function. The linear search allows arriving at a point below the extruded surface intersection with the view ray. The following binary search helps finding an approximate height field intersection utilizing bilinear texture filtering to interpolate the intersection point.

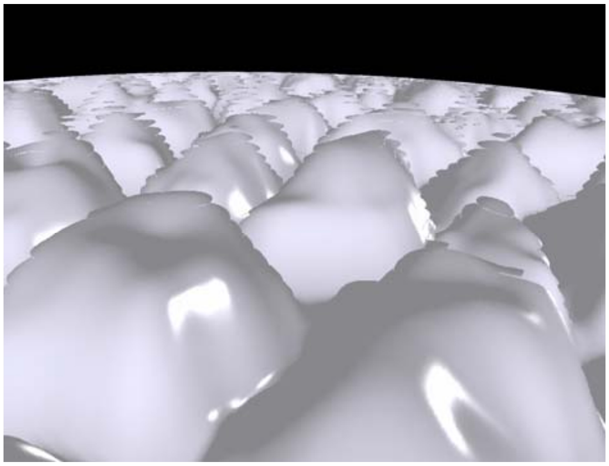


***Figure 4a**. Relief mapping rendered with both linear and binary search but without depth bias applied. Notice the visual artifacts due to sampling aliasing at grazing angles.*

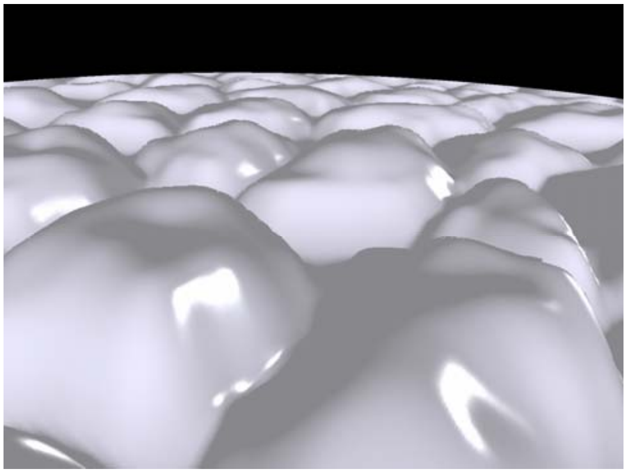


***Figure 4b**. Relief mapping rendered with both linear and binary search and with depth bias applied. Notice the flattening of surface features towards the horizon.*

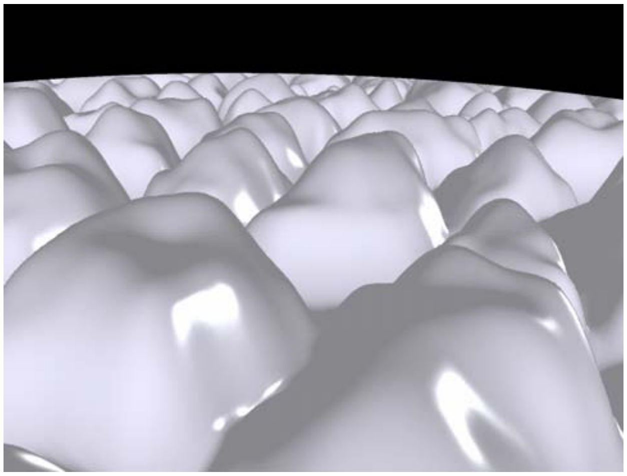
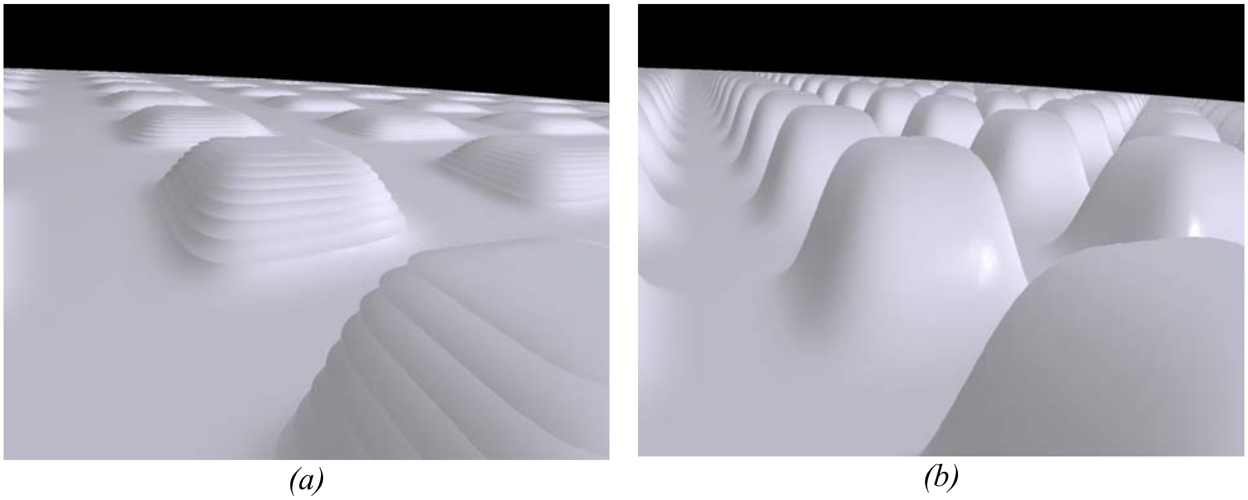


Figure 4c. *Parallax occlusion mapping rendered with the high precision height field intersection computation. Notice the lack of aliasing artifacts or feature flattening toward the horizon.*

The intersection of the surface is approximated with texture filtering, thus only using 8 bits of precision for the intersection computation. This results in visible stair-stepping artifacts at steep viewing angles (as seen in Figure 4a). Depth biasing toward the horizon hides these artifacts but introduces excessive feature flattening at oblique angles (Figure 4b).

The binary search from [Policarpo et al. 2005] requires dependent texture fetches for computation. These incur a latency cost which is not offset by any ALU computations in the relief mapping ray-height field intersection routine. Increasing the sampling rate during the binary search increases the latency of each fetch by increasing the dependency depth for each successive fetch.

Using a linear search from [Policarpo et al. 2005] without an associated binary search exacerbates the stair-stepping artifacts even with a high sampling rate (as in Figure 5a).



(a) *(b)*

Figure 5. *Comparison of height field intersection precision using the linear search only (same assets). (a) Relief mapping. (b) Parallax occlusion mapping*

The advantage of a linear search for intersection root finding lies in an effective use of texturing hardware with low latency as it does not require dependent texture fetches. Simply using a linear search requires higher precision for the root-finding.

We sample the height field using a linear search and approximating the height profile as a piecewise linear curve (as illustrated in Figure 6). This allows us to combine the 8 bit precision due to bilinear texture filtering with the full 32 bit precision for root finding during the line intersection. Figure 4c displays the improved visual results with the lack of aliasing with using our approach.

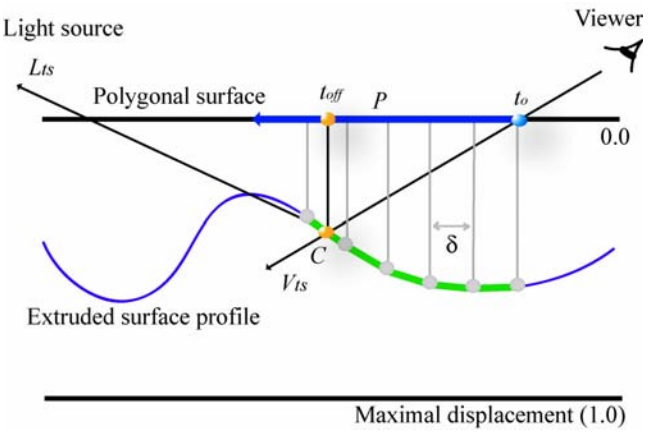


***Figure 6**. We start at the input texture coordinates t_o and sample the height field profile for each linear segment of the green piecewise linear curve along parallax offset vector P. The height field profile-view ray intersection yields parallax-shifted texture coordinate offset t_{off}. δ is the interval step size. Then we perform the visibility tracing. We start at texture offset t_{off} and trace along the light direction vector L_{ts} to determine any occluding features in the height field profile.*

Since we do not encode feature information into additional look-up tables, the accuracy of our technique corresponds to the sampling interval δ (as well as for [Policarpo et al. 2005]). Both algorithms suffer from some amount of aliasing artifacts if too few samples are used for a relatively high-frequency height field, though the amount will differ between the techniques.

Automatically determining δ by using the texture resolution is currently impractical. At grazing angles, the parallax amount is quite large and thus we must march along a long parallax offset vector in order to arrive at the actual displaced point. In that case, the step size is frequently much larger than a texel, and thus unrelated to the texture resolution. To solve this, we provide both directable and automatic controls.

The artists can control the sampling size bounds with artist-editable parameters. This is convenient in real game scenarios as it allows control per texture map. If dynamic flow control (DFC) is available, we can automatically adjust the sampling rate during ray tracing. We express the sampling rate as a linear function of the angle between the geometric normal and the view direction ray:

$$n = n_{\min} + \hat{N} \bullet \hat{V}_{ts}(n_{\max} - n_{\min}) \quad (1)$$

where $n_{\min}$ and $n_{\max}$ are the artist-controlled sampling rate bounds, $\hat{N}$ is the interpolated geometric unit normal vector at the current pixel. This assures that we increase the sampling rate along the steep viewing angles. We increase the efficiency of the linear search by using the early out functionality of DFC to stop sampling the height field when a point below the surface is found.

5.4.2 Soft Shadows

The height map can in fact cast shadows on itself. This is accomplished by substituting the light vector for the view vector when computing the intersection of the height profile to determine the correct displaced texel position during the reverse height mapping step. Once we arrive at the point on the displaced surface (the point *C* in figure 6) we can compute its visibility from the any light source. For that, we cast a ray toward the light source in question and perform horizon visibility queries of the height field profile along the light direction ray $\hat{L}_{ts}$.

If there are intersections of the height field profile with $\hat{L}_{ts}$, then there are occluding features and the point in question will be in shadow. This process allows us to generate shadows due to the object features' self-occlusions and object interpenetration.

If we repeated the process for the height field profile – view direction ray tracing for the visibility query by stopping sampling at the very first intersection, we would arrive at the horizon shadowing value describing whether the displaced pixel is in shadow. Using this value during the lighting computation (as in [Policarpo et al. 2005]) generates hard shadows which can display aliasing artifacts in some scenes (Figure 7a).

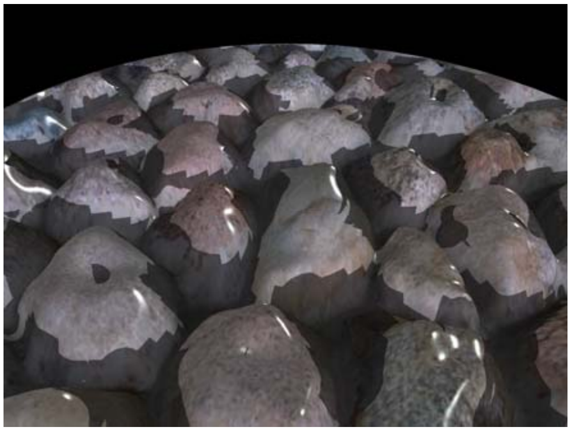


***Figure 7a**. Hard shadows generated with the relief mapping horizon visibility threshold computation.*

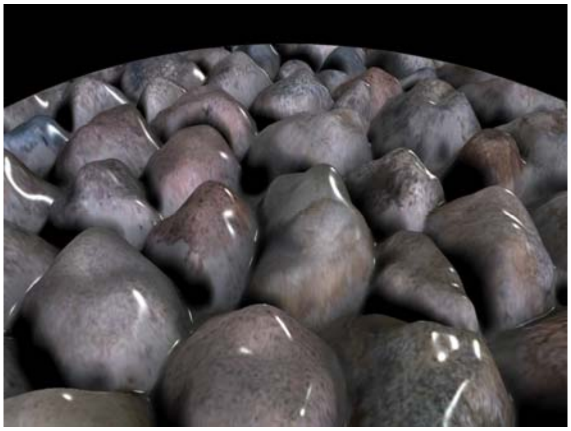


***Figure 7b.** Soft shadows generated with the parallax occlusion mapping penumbra approximation technique.*

With our approach, we sample n samples h_1 - h_n along the light direction ray (Figure 8). We assume that we are lighting the surface with an area light source and, similar to [Chan and Durand 2003] and [Wyman and Hansen 2003], we heuristically approximate the size of penumbra for the occluded pixel. Figure 9 demonstrates the penumbra size computation given an area light source, a blocker and a receiver surface.

We can use the height field profile samples h_i along the light direction ray to determine the occlusion coefficient. We sample the height value h_0 at the shifted texture coordinate t_{off}. Starting at this offset ensures that the shadows do not appear floating on top of the surface. The sample h_0 is our reference ("surface") height. We then sample n other samples along the light ray, subtracting h_0 from each of the successive samples h_i. This allows us to compute the blocker-to-receiver ratio as in figure 9. The closer the blocker is to the surface, the smaller the resulting penumbra. We compute the penumbra coefficient (the visibility coefficient v) by scaling the contribution of each sample by the distance of this sample from h_0, and using the maximum value sampled. Additionally we can weight each visibility sample to simulate the blur kernel for shadow filtering.

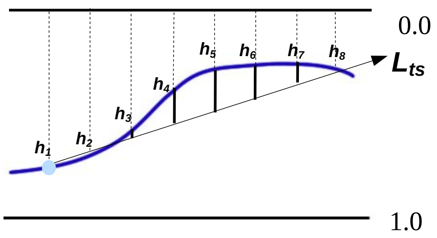


Figure 8. *Sampling the height field profile along the light ray direction L_{ts} to obtain height samples $h_1 - h_8$ (n=8)*

$$w_p = \frac{w_s(d_r - d_b)}{d_b}$$

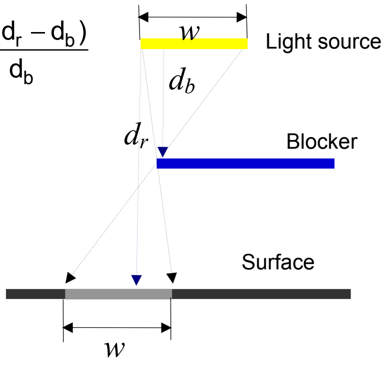


Figure 9. *Penumbra size approximation for area light sources, where w_s is the light source width, w_p is the penumbra width, d_r is the receiver depth and d_b is the blocker depth from the light source.*

We apply the visibility coefficient v during the lighting computation for generation of smooth soft shadows. This allows us to obtain well-behaved soft shadows without any edge aliasing or filtering artifacts. Figures 7b and 10 demonstrate examples of smooth shadows using our technique. One limitation of our technique is the lack of hard surface contact shadow for extremely high frequency height maps.

Remember that estimating light visibility increases shader complexity. We perform the visibility query only for areas where the dot product between the geometric normal and the light vector is non-negative by utilizing dynamic branching (see the actual pixel shader in Listing 2 in the Appendix). This allows us to compute soft shadows only for areas which are actually facing the light source.

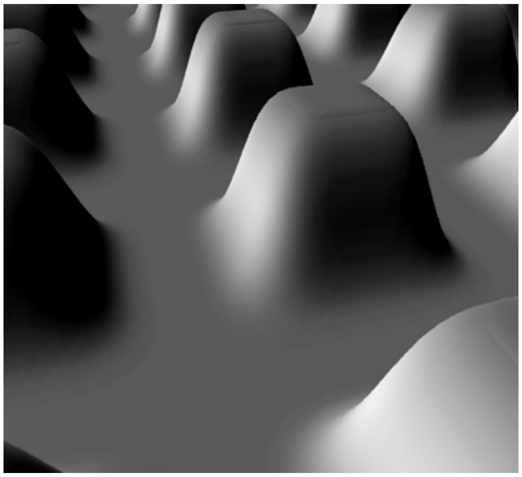


Figure 10. *Smooth soft shadows and perspective-correct depth details generated with the parallax occlusion rendering algorithm*

5.4.3 Adaptive Level-of-Detail Control System

We compute the current mip level explicitly in the pixel shader and use this information to transition between different levels of detail from the full effect to bump mapping. Simply using mip-mapping for LOD management is ineffective since it does not reduce shader complexity during rendering. Using the full shader for the height field profile intersection with the view ray and the light ray, the visibility and lighting is expensive. Although at lower mip levels the fill is reduced, without our level-of-detail system, the full shader will be executed for each pixel regardless of its proximity to the viewer. Instead, with our algorithm, only a simple bump mapping shader is executed for mip levels higher than the specified threshold value.

This in-shader LOD system provides a significant rendering optimization and smooth transitions between the full parallax occlusion mapping and a simplified representation without visual artifacts such as ghosting or popping. Since all calculations are performed per pixel, the method robustly handles extreme close-ups of the object surface, thus providing an additional level of detail management.

We compute the mip level directly in the pixel shader (as described in [Shreiner et al. 2005]) on SM 3.0 hardware (see the actual pixel shader in Listing 2 in the Appendix). The lowest level of detail is rendered using bump mapping. As we determine that the currently rendered level of detail is close enough to the threshold, we interpolate the parallax occlusion-mapped lit result with the bump-mapped lit result using the fractional part of the current mip level as the interpolation parameter. There is almost no associated visual quality degradation as we move into a lower level of detail and the transition appears quite smooth (Figure 11).

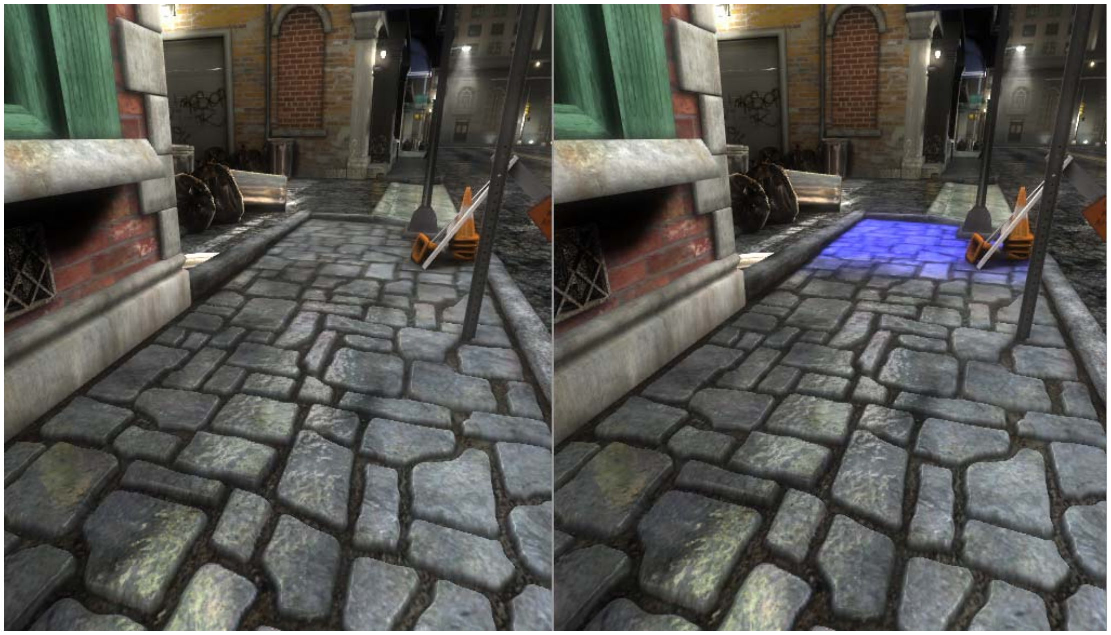


Figure 11. *A complex scene with parallax occlusion mapping on the sidewalk and the brick wall. The strength of the blue tint in (b) denotes the decreasing level of detail (deepest blue being bump mapping and no blue displays full computation). Note the lack of visual artifacts and smooth transition due to the level-of-detail transition discernable in (a). The transition region is very small.*

We expose the threshold level parameter to the artists in order to provide directability for game level editing. In our examples we used a threshold value of 4. Thus even at steep grazing angles the close-up views of surfaces will maintain perspective correct depth.

5.5 Results

We have implemented the techniques described in this paper using DirectX 9.0c shader programs on a variety of graphics cards. An example of an educational version of this shader is shown in listings 1 (for the vertex shader implementation using DirectX 9.0c shader model 3.0) and listing 2 (for the pixel shader implementation using DirectX 9.0c shader model 3.0). We use different texture sizes selected based on the desired feature resolution. For Figures 1, 2, 11, 16, and 17 we apply 1024x1024 RGBα textures with non-contiguous texture coordinates. For Figures 4 and 7 we apply repeated 256x256 RGBα textures, and for Figures 5 and 10 we use repeated 128x128 RGBα textures.

We applied parallax occlusion mapping to a 1,100 polygon soldier character shown in figure 12a. We compared this result to a 1.5 million polygon version of the soldier model used to generate normal maps for the low resolution version (Figure 12b). (Note that the two images in figure 12 are from slightly different viewpoints though extracted from a demo sequence with similar viewpoint paths.) We apply a 2048x2048 RGBα texture map to the low resolution object. We render the low resolution soldier using DirectX on ATI

Radeon X1600 XL at 255 fps. The sampling rate bounds were set to the range of [8, 50] range. The memory requirement for this model was 79K for the vertex buffer, 6K for the index buffer, and 13MB of texture memory (using 3DC texture compression).

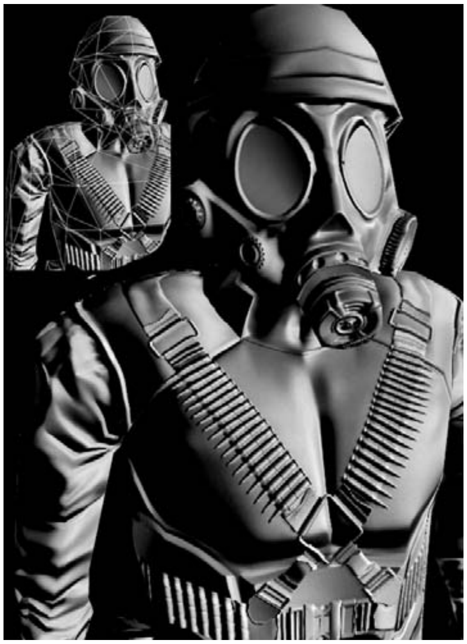


Figure 12a. *An 1,100 polygon game soldier character with parallax occlusion mapping*

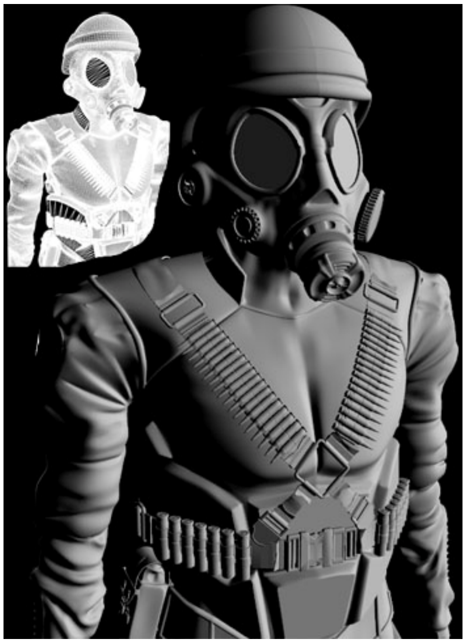


Figure 12b. *A 1.5 million polygon soldier character with diffuse lighting*

The high resolution soldier model is rendered on the same hardware at a rate of 32 fps. The memory requirement for this model was 31MB for the vertex buffer and 14MB for the index buffer. Due to memory considerations, vertex transform cost for rendering, animation, and authoring issues, characters matching the high resolution soldier are impractical in current game scenarios. However, using our technique on an extremely low resolution model provided significant frame rate increase with 32MB of memory being saved, at a comparable quality of rendering.

This demonstrates the usefulness of the presented technique for texture-space displacement mapping via parallax occlusion mapping. In order to render the same objects interactively with equal level of detail, the meshes would need an extremely detailed triangle subdivision, which is impractical even with the currently available GPUs.

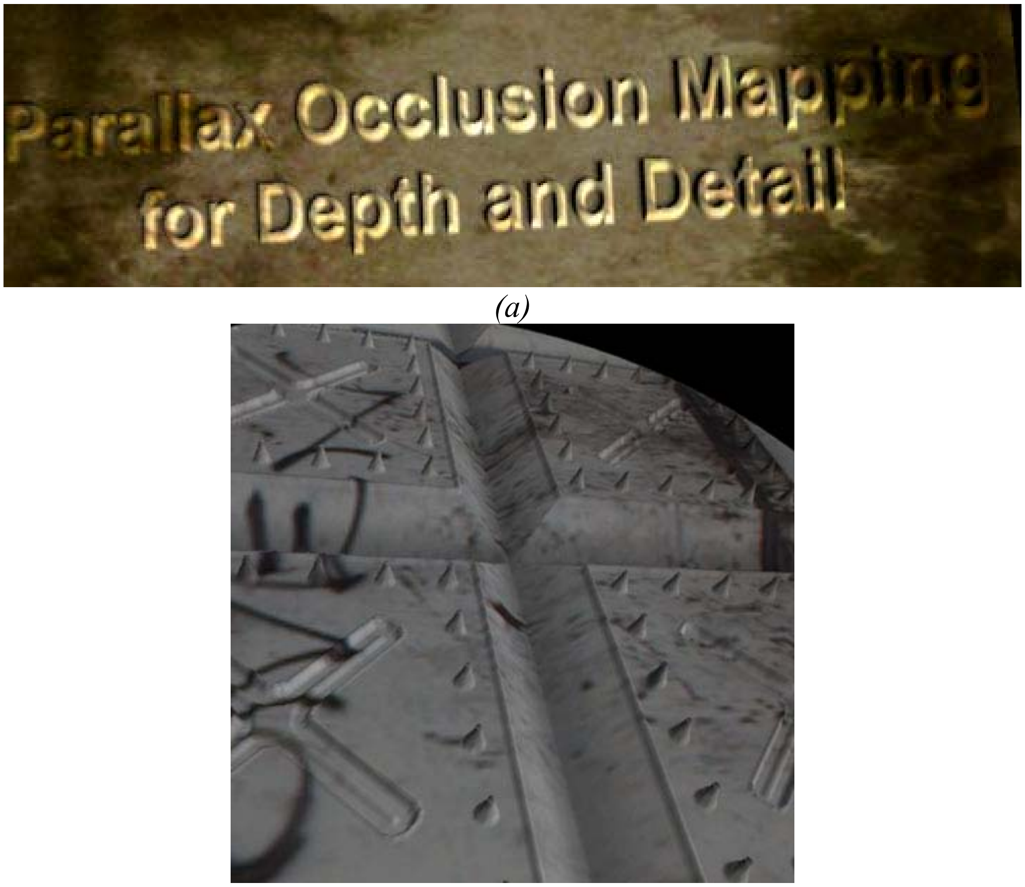
Our method can be used with a dynamically rendered height field and still produce perspective-correct depth results. In that case, the dynamically updated displacement values can be used to derive the normal vectors at rendering time by convolving the height map with a Sobel operator in the horizontal and vertical direction (as described in detail in [Tatarchuk06a]). The rest of the algorithm does not require any modification.

We used this technique extensively in the interactive demo called “ToyShop” [Toyshop05] for a variety of surfaces and effects. As seen in Figure 1 and 16, we’ve rendered the cobblestone sidewalk using this technique (using sampling range from 8 to 40 samples per pixel), in Figure 13 we have applied it to the brick wall (with the same sampling range), and in Figure 17 we see parallax occlusion mapping used to render extruded wood-block letters of the ToyShop store sign. We were able to integrate a variety of lighting models with this technique, ranging from a single diffusely lit material in Figure 13, to shadows and specular illumination in Figure 17, and shadow mapping integrated and dynamic view-dependent reflections in Figure 1a.

5.6 Considerations for practical use of parallax occlusion mapping and game integration

5.6.1 Algorithm limitations and relevant considerations

Although parallax occlusion mapping is a very powerful and flexible technique for computing and lighting extruded surfaces in real-time, it does have its limitations. Parallax occlusion mapping is a sampling-based algorithm at its core, and as such, it can exhibit aliasing. The frequencies of the height field will determine the required sampling rate for the ray tracing procedures – otherwise aliasing artifacts will be visible (as seen in Figures 4a and 5a). One must increase the sampling rate significantly if the height field contains very sharp features (as visible in the text and sharp conic features in Figure 13 below). However, as we can note from the images in Figures 13a and 13b, the visual quality of the results rendered with parallax occlusion mapping is high enough to render such traditionally difficult objects as extruded text or sharp peaks at highly interactive rates (fps > 15fps on ATI Radeon X1600 XL rendering at 1600x1200 resolution). In order to render the same objects interactively with equal level of detail, the meshes would need an extremely detailed triangle subdivision (with triangles being nearly pixel-sized), which is impractical even with the currently available GPUs.



(a)

(b)

Figure 13. *Rendering extruded text objects in (a) and sharp conic features in (b) with parallax occlusion mapping*

The sampling limitation is particularly evident in the DirectX 9.0c shader model 2.0 implementation of the parallax occlusion mapping algorithm if the height field used has high spatial frequency content. This specific shader model suffers from a small instruction count limit, and thus we are unable to compute more than 8 samples during ray tracing in a single pass. However, several passes can be used to compute the results of ray tracing by using offscreen buffer rendering to increase the resulting precision of computations using SM 2.0 shaders. As in the analog-to-digital sound conversion process, sampling during the ray tracing at slightly more than twice the frequency of the height map features will make up for not modeling the surfaces with implicit functions and performing the exact intersection of the ray with the implicit representation of the extruded surface.

Another limitation of our technique is the lack of detailed silhouettes, betraying the low resolution geometry actually rendered with our method. This is an important feature for enhancing the realism of the effect and we are investigating ideas for generating correct silhouettes. However, in many scenarios, the artists can work around this issue by placing specific ‘border’ geometry to hide the artifacts. One can notice this at work in the “ToyShop” demo as the artists placed the curb stones at the edge of the sidewalk object with parallax occlusion mapped cobblestones or with a special row of bricks at the corner of the brick building seen in Figure 14 below.

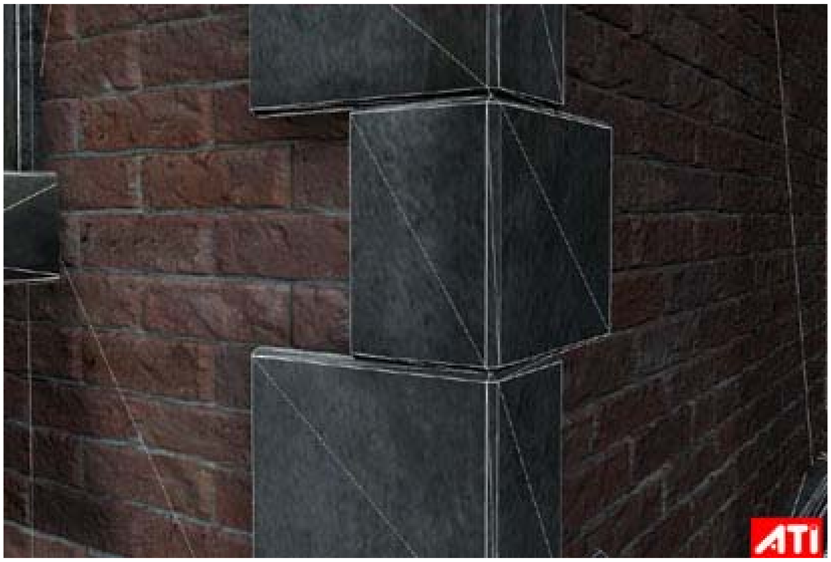


Figure 14. *Additional brick geometry placed at the corners of parallax occlusion mapped brick objects to hide inaccurate silhouettes in an interactive environment of the "ToyShop" demo.*

The parallax occlusion mapping algorithm will not automatically produce surfaces with correct depth buffer values (since it simply operates in screen-space on individual pixels). This means that in some situations this will result in apparent object interpenetration or incorrect object collision. The algorithm can be extended to output accurate depth quite easily. Since we know the reference surface's geometric depth, we can compute the displacement amount by sampling the height field at the t_{off} location and adding or subtracting this displacement amount to the reference depth value (as described in [Policarpo05]) by outputting it as the depth value from the pixel shader.

5.6.1 Art Content Authoring Suggestions for Parallax Occlusion Mapping

Adding art asset support for parallax occlusion mapping requires a minimal increase in memory footprint (for an additional 8-bit height map) if the application already supports normal mapping and contains appropriate assets. There are many reliable methods for generating height maps useful for this technique:

- Normal maps can be generated from a combination of a low- and high-resolution models with the *NormalMapper* software [NormalMapper03]. The tool has an option to simultaneously output the corresponding displacement values in a height map
- A height map may be painted in a 3D painting software like ZBrush^{TM}
- It also can be created in a 2D painting software such as Adobe® Photoshop^{TM}

The parallax occlusion mapping technique is an efficient and compelling technique for simulating surface details. However, as with other bump mapping techniques, its quality depends strongly on the quality of its art content. Empirically, we found that lower-frequency height map textures result in higher performance (due to less samples required for ray tracing) and better quality results (since less height field features are missed). For example, if creating height maps for rendering bricks or cobblestones, one may widen the grout region and apply a soft blur to smooth the transition and thus lower the height map frequency content. As discussed in the previous section, when using high frequency height maps (such as those in Figure 13a or 13b), we must increase the range of sampling for ray tracing.

An important consideration for authoring art assets for use with this algorithm lies in the realization that the algorithm always extrudes surfaces “pushing down” – unlike the traditional displacement mapping. This affects the placement of the original low resolution geometry – the surfaces must be placed slightly higher than where the anticipated extruded surface should be located. Additionally this means that the peaks in the extruded surface will correspond to the brightest values in the height map (white) and the valleys will be corresponding to the darkest (black).

5.7 Conclusions

We have presented a pixel-driven displacement mapping technique for rendering detailed surfaces under varying light conditions, generating soft shadows due to self-occlusion. We have described an efficient algorithm for computing intersections of the height field profile with rays with high precision. Our method includes estimation of light visibility for generation of soft shadows. An automatic level-of-detail control system manages shader complexity efficiently at run-time, generating smooth LOD transitions without visual artifacts. Our technique takes advantage of the programmable GPU pipeline resulting in highly interactive frame rates coupled with a low memory footprint.

Parallax occlusion mapping can be used effectively to generate an illusion of very detailed geometry exhibiting correct motion parallax as well as producing very convincing self-shadowing effects. We provide a high level of directability to the artists and significantly improved visual quality over the previous approaches. We hope to see more games implementing compelling scenes using this technique.

5.7 Acknowledgements

We would like to thank Dan Roeger, Daniel Szecket, Abe Wiley and Eli Turner for their help with the artwork, and Zoë Brawley from Relic Entertainment for her ideas in the original implementation of the 2004 technique. We also would like to thank Pedro Sander, John Isidoro, Thorsten Scheuermann, and Chris Oat of ATI Research, Inc., Jason L. Mitchell of Valve Software, and Eric Haines, of Autodesk for their help, suggestions and review of this work.

5.8 Bibliography

BECKER, B. G., AND MAX, N. L. 1993. Smooth Transitions between Bump Rendering Algorithms. In ACM Transactions on Graphics (Siggraph 1993 Proceedings), ACM Press, pp. 183-190

BLINN, J. F. 1978. “Simulation of Wrinkled Surfaces”. In Proceedings of the 5th annual conference on Computer graphics and interactive techniques, ACM Press, pp. 286-292.

BRAWLEY, Z., AND TATARCHUK, N. 2004. Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing. In ShaderX^3: Advanced Rendering with DirectX and OpenGL, Engel, W., Ed., Charles River Media, pp. 135-154.

CHAN, E., AND DURAND, F. 2003. Rendering fake soft shadows with smoothies, In Eurographics Symposium on Rendering Proceedings, ACM Press, pp. 208-218.

COOK, R. L. 1984. Shade Trees, In Proceedings of the 11th annual conference on Computer graphics and interactive techniques, ACM Press, pp. 223-231.

DOGGETT, M., AND HIRCHE, J. 2000. Adaptive View Dependent Tessellation of Displacement Maps. In HWWS ’00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware, ACM Press, pp. 59-66.

DONNELLY, W. 2005. Per-Pixel Displacement Mapping with Distance Functions. In GPU Gems 2, M. Pharr, Ed., Addison-Wesley, pp. 123 – 136.

FOURNIER, A. 1992. Filtering Normal Maps and Creating Multiple Surfaces, Technical Report, University of British Columbia.

HEIDRICH, W., AND SEIDEL, H.-P. 1998. Ray-tracing Procedural Displacement Shaders, In Graphics Interface, pp. 8-16.

HIRCHE, J., EHLERT, A., GUTHE, S., DOGGETT, M. 2004. Hardware Accelerated Per-Pixel Displacement Mapping. In Graphics Interface, pp. 153-158.

KANEKO, T., TAKAHEI, T., INAMI, M., KAWAKAMI, N., YANAGIDA, Y., MAEDA, T., TACHI, S. 2001. Detailed Shape Representation with Parallax Mapping. In Proceedings of ICAT 2001, pp. 205-208.

KAUTZ, J., AND SEIDEL, H.-P. 2001. Hardware accelerated displacement mapping for image based rendering. In Proceedings of Graphics Interface 2001, B.Watson and J.W. Buchanan, Eds., pp. 61–70.

MAX, N. 1988. Horizon mapping: shadows for bump-mapped surfaces. The Visual Computer 4, 2, pp. 109–117.

MCGUIRE, M. AND MCGUIRE, M. 2005. Steep Parallax Mapping. I3D 2005 Poster.

AKENINE-MÖLLER, T., HEINES, E. 2002. *Real-Time Rendering*, 2nd Edition, A.K. Peters, July 2002

OLANO, M., KUEHNE, B., SIMMONS, M. 2003. Automatic Shader Level of Detail. In Siggraph/Eurographics Workshop on Graphics Hardware Proceedings, ACM Press, pp. 7-14.

OLIVEIRA, M. M, AND POLICARPO, F.. 2005. An Efficient Representation for Surface Details. UFRGS Technical Report RP-351.

OLIVEIRA, M. M., BISHOP, G., AND MCALLISTER, D. 2000. Relief texture mapping. In Siggraph 2000, Computer Graphics Proceedings, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., pp. 359–368.

PHARR, M., AND HANRAHAN, P. 1996. Geometry caching for ray-tracing displacement maps. In Eurographics Rendering Worshop 1996, Springer Wien, New York City, NY, X. Pueyo and P. Schröder, Eds., pp. 31–40.

POLICARPO, F., OLIVEIRA, M. M., COMBA, J. 2005. Real-Time Relief Mapping on Arbitrary Polygonal Surfaces. In ACM SIGGRAPH Symposium on Interactive 3D Graphics Proceedings, ACM Press, pp. 359-368.

POLICARPO, F., OLIVEIRA, M. M. 2006. Relief Mapping of Non-Height-Field Surface Details. In ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games Proceedings, ACM Press, pp. 55-52.

SHREINER, D., WOO, M., NEIDER, J., DAVIS, T.. 2005. OpenGL® Programming Guide: The Official Guide to Learning OpenGL®, version 2, Addison-Wesley.

SLOAN, P-P. J., AND COHEN, M. F. 2000. Interactive Horizon Mapping. In 11th Eurographics Workshop on Rendering Proceedings, ACM Press, pp. 281-286.

TATARCHUK, N. 2006. Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows. In the proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, pp. 63-69

TATARCHUK, N. 2006a. Practical Parallax Occlusion Mapping for Highly Detailed Surface Rendering. In the proceedings of Game Developer Conference

WANG, L., WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H.-Y. 2003. View-dependent displacement mapping. ACM Trans. Graph. 22, 3, pp. 334–339.

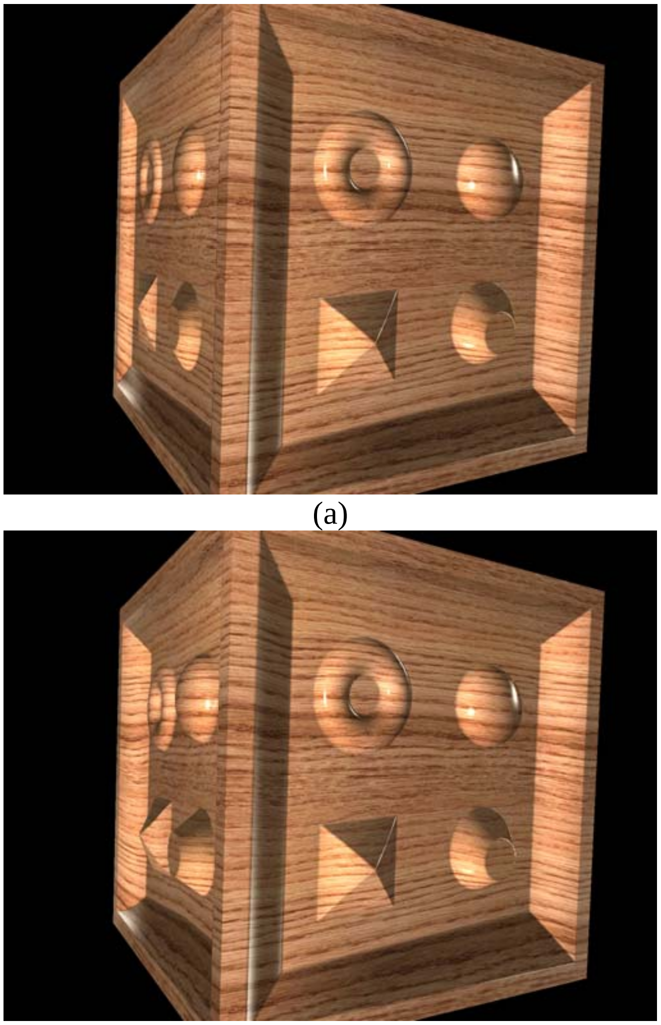
WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H.-Y. 2004. Generalized displacement maps. In Eurographics Symposium on Rendering 2004, EUROGRAPHICS, Keller and Jensen, Eds., EUROGRAPHICS, pp. 227–233.

WELSH, T. 2004. Parallax Mapping, ShaderX3: Advanced Rendering with DirectX and OpenGL, Engel, W. Ed., A.K. Peters, 2004

WYMAN, C., AND HANSEN, C. 2002. Penumbra maps: approximate soft shadows in real-time. In Eurographics workshop on Rendering 2003, EUROGRAPHICS, Keller and Jensen, Eds., EUROGRAPHICS, pp. 202-207.

TOYSHOP DEMO, 2005. ATI Research, Inc. Can be downloaded from http://www.ati.com/developer/demos/rx1800.html

NORMALMAPPER TOOL, 2003. ATI Research, Inc. Can be downloaded from http://www2.ati.com/developer/NormalMapper-3_2_2.zip



(a)

(b)

Figure 15. *Simple cube model rendered with detailed surface detailed from the same viewpoint. In (a), relief mapping is used to create surface complexity. In (b), parallax occlusion mapping is used to render perspective-correct extruded surfaces. Notice the differences on the left face of the cube as the surface is viewed at a steep angle.*

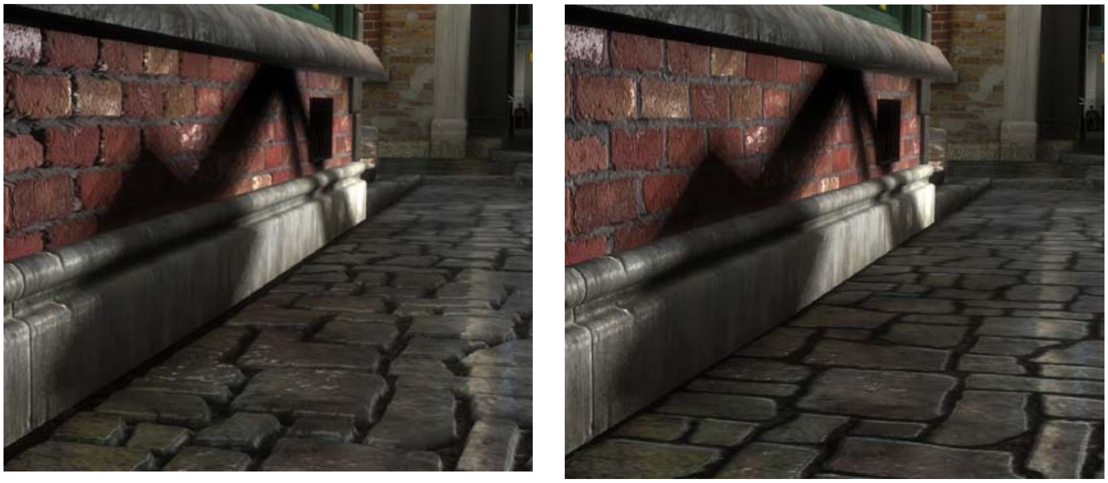


Figure 16. *A portion of a realistic city environment with the cobblestone sidewalk and the brick wall rendered with parallax occlusion mapping (left) and bump mapping (right). We are able to use shadow mapping on the surfaces and dynamically rendered reflections from objects in the scene.*

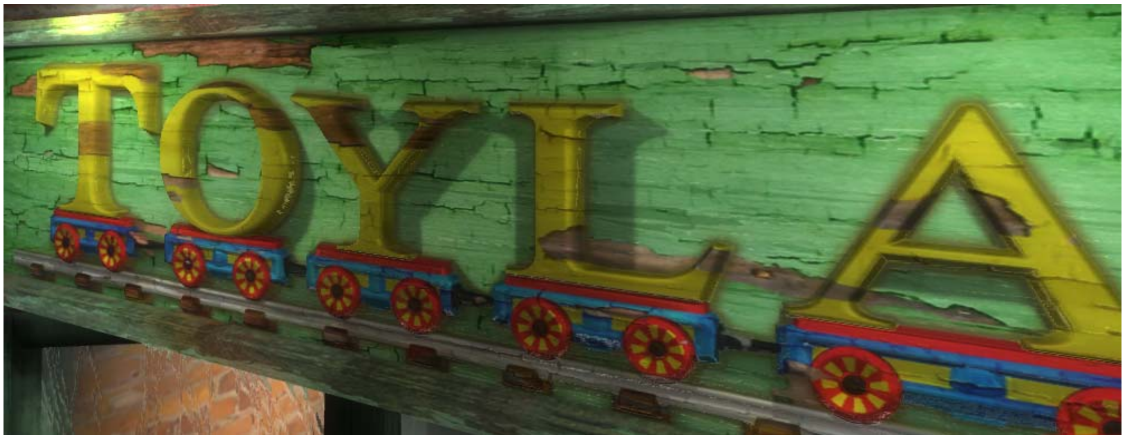


Figure 17. *Displaced text rendering with the sign rendered using parallax occlusion mapping technique*

Appendix. DirectX shader code implementation of Parallax Occlusion Mapping.

```
float4x4 matViewInverse;
float4x4 matWorldViewProjection;
float4x4 matView;

float    fBaseTextureRepeat;
float    fHeightMapRange;
float4   vLightPosition;

struct VS_INPUT
{
   float4 positionWS  : POSITION;
   float2 texCoord    : TEXCOORD0;
   float3 vNormalWS   : NORMAL;
   float3 vBinormalWS : BINORMAL;
   float3 vTangentWS  : TANGENT;
};

struct VS_OUTPUT
{
   float4 position : POSITION;
   float2 texCoord : TEXCOORD0;

   // Light vector in tangent space, not normalized
   float3 vLightTS : TEXCOORD1;

   // View vector in tangent space, not normalized
   float3 vViewTS : TEXCOORD2;

   // Parallax offset vector in tangent space
   float2 vParallaxOffsetTS : TEXCOORD3;

   // Normal vector in world space
   float3 vNormalWS : TEXCOORD4;

   // View vector in world space
   float3 vViewWS : TEXCOORD5;
};

VS_OUTPUT vs_main( VS_INPUT i )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;

   // Transform and output input position
   Out.position = mul( matWorldViewProjection, i.positionWS );

   // Propagate texture coordinate through:
   Out.texCoord = i.texCoord;

   // Uncomment this to repeat the texture
   // Out.texCoord *= fBaseTextureRepeat;

   // Propagate the world vertex normal through:
```

```
   Out.vNormalWS = i.vNormalWS;

   // Compute and output the world view vector:
   float3 vViewWS = mul( matViewInverse,
                    float4(0,0,0,1)) - i.positionWS;

   Out.vViewWS = vViewWS;

   // Compute denormalized light vector in world space:
   float3 vLightWS = vLightPosition - i.positionWS;

   // Normalize the light and view vectors and transform
   // it to the tangent space:
   float3x3 mWorldToTangent =
      float3x3( i.vTangentWS, i.vBinormalWS, i.vNormalWS );

   // Propagate the view and the light vectors (in tangent space):
   Out.vLightTS = mul( mWorldToTangent, vLightWS );
   Out.vViewTS  = mul( mWorldToTangent, vViewWS  );

   // Compute the ray direction for intersecting the height field
   // profile with current view ray. See the above paper for derivation
   // of this computation.

   // Compute initial parallax displacement direction:
   float2 vParallaxDirection = normalize(  Out.vViewTS.xy );

   // The length of this vector determines the furthest amount
   // of displacement:
   float fLength = length( Out.vViewTS );
   float fParallaxLength = sqrt( fLength * fLength - Out.vViewTS.z
            * Out.vViewTS.z ) / Out.vViewTS.z;

   // Compute the actual reverse parallax displacement vector:
   Out.vParallaxOffsetTS = vParallaxDirection * fParallaxLength;

   // Need to scale the amount of displacement to account for
   // different height ranges in height maps. This is controlled by
   // an artist-editable parameter:
   Out.vParallaxOffsetTS *= fHeightMapRange;

   return Out;

}   // End of VS_OUTPUT vs_main(..)
```

Listing 1. *Parallax occlusion mapping algorithm implementation. Vertex shader, DirectX 9.0c shader model 3.0*

```
// NOTE: Since for this particular example want to make convenient ways
// to turn features rendering on and off (for example, for turning on /
// off visualization of current level of details, shadows, etc), the
// shader presented uses extra flow control instructions than it would
// in a game engine.

// Uniform shader parameters declarations
bool bVisualizeLOD;
bool bVisualizeMipLevel;
bool bDisplayShadows;

// This parameter contains the dimensions of the height map / normal map
// pair and is used for determination of current mip level value:
float2 vTextureDims;

int   nLODThreshold;
float fShadowSoftening;
float fSpecularExponent;
float fDiffuseBrightness;
float fHeightMapRange;

float4 cAmbientColor;
float4 cDiffuseColor;
float4 cSpecularColor;

int nMinSamples;
int nMaxSamples;

sampler tBaseMap;
sampler tNormalMap;

// Note: centroid sampling should be specified if multisampling is
// enabled
struct PS_INPUT
{
   float2 texCoord : TEXCOORD0;

   // Light vector in tangent space, denormalized
   float3 vLightTS : TEXCOORD1_centroid;

   // View vector in tangent space, denormalized
   float3 vViewTS : TEXCOORD2_centroid;

   // Parallax offset vector in tangent space
   float2 vParallaxOffsetTS : TEXCOORD3_centroid;

   // Normal vector in world space
   float3 vNormalWS : TEXCOORD4_centroid;

   // View vector in world space
   float3 vViewWS : TEXCOORD5_centroid;
};
```

```
//......................................................................
// Function:    ComputeIllumination
//
// Description: Computes phong illumination for the given pixel using
//              its attribute textures and a light vector.
//......................................................................
float4 ComputeIllumination( float2 texCoord, float3 vLightTS,
                            float3 vViewTS,  float  fOcclusionShadow )
{
   // Sample the normal from the normal map for the given texture sample:
   float3 vNormalTS = normalize( tex2D( tNormalMap, texCoord ) * 2 - 1 );

   // Sample base map:
   float4 cBaseColor = tex2D( tBaseMap, texCoord );

   // Compute diffuse color component:
   float4 cDiffuse = saturate( dot( vNormalTS, vLightTS )) *
                               cDiffuseColor;

   // Compute specular component:
   float3 vReflectionTS = normalize( 2 * dot( vViewTS, vNormalTS ) *
                                     vNormalTS - vViewTS );

   float fRdotL = dot( vReflectionTS, vLightTS );

   float4 cSpecular = saturate( pow( fRdotL, fSpecularExponent )) *
                      cSpecularColor;

   float4 cFinalColor = (( cAmbientColor + cDiffuse ) * cBaseColor +
                      cSpecular ) * fOcclusionShadow;

   return cFinalColor;
}

//.......................................................................
// Function:    ps_main
//
// Description: Computes pixel illumination result due to applying
//              parallax occlusion mapping to simulation of view-
//              dependent surface displacement for a given height map
//.......................................................................
float4 ps_main( PS_INPUT i ) : COLOR0
{
   //  Normalize the interpolated vectors:
   float3 vViewTS   = normalize( i.vViewTS  );
   float3 vViewWS   = normalize( i.vViewWS  );
   float3 vLightTS  = normalize( i.vLightTS );
   float3 vNormalWS = normalize( i.vNormalWS );

   float4 cResultColor = float4( 0, 0, 0, 1 );

   // Adaptive in-shader level-of-detail system implementation.
   // Compute the current mip level explicitly in the pixel shader
   // and use this information to transition between different levels
   // of detail from the full effect to simple bump mapping.

   // Compute the current gradients:
   float2 fTexCoordsPerSize = i.texCoord * vTextureDims;

   // Compute all 4 derivatives in x and y in a single instruction
   //  to optimize:
   float2 dxSize, dySize;
   float2 dx, dy;

   float4( dxSize, dx ) = ddx( float4( fTexCoordsPerSize, i.texCoord ) );
   float4( dySize, dy ) = ddy( float4( fTexCoordsPerSize, i.texCoord ) );

   float  fMipLevel;
   float  fMipLevelInt;    // mip level integer portion
   float  fMipLevelFrac;   // mip level fractional amount for
                           // blending in between levels

   float  fMinTexCoordDelta;
   float2 dTexCoords;

   // Find min of change in u and v across quad: compute du and dv
   // magnitude across quad
   dTexCoords = dxSize * dxSize + dySize * dySize;

   // Standard mipmapping uses max here
   fMinTexCoordDelta = max( dTexCoords.x, dTexCoords.y );

   // Compute the current mip level  (* 0.5 is effectively
   // computing a square root before )
   fMipLevel = max( 0.5 * log2( fMinTexCoordDelta ), 0 );

   // Start the current sample located at the input texture
   // coordinate, which would correspond to computing a bump
   // mapping result:
   float2 texSample = i.texCoord;

   // Multiplier for visualizing the level of detail
   float4 cLODColoring = float4( 1, 1, 3, 1 );

   float fOcclusionShadow = 1.0;

   if ( fMipLevel <= (float) nLODThreshold )
   {
      //===============================================//
      // Parallax occlusion mapping offset computation //
      //===============================================//

      // Utilize dynamic flow control to change the number of samples
      // per ray depending on the viewing angle for the surface.
      // Oblique angles require smaller step sizes to achieve
      // more accurate precision for computing displacement.
      // We express the sampling rate as a linear function of the
      // angle between the geometric normal and the view direction ray:
      int nNumSteps = (int) lerp( nMaxSamples, nMinSamples,
                                  dot( vViewWS, vNormalWS ) );

      // Intersect the view ray with the height field profile along
      // the direction of the parallax offset ray (computed in the
      // vertex shader. Note that the code is designed specifically
```

```
// to take advantage of the dynamic flow control constructs in HLSL
// and is very sensitive to the specific language syntax.
// When converting to other examples, if still want to use dynamic
// flow control in the resulting assembly shader, care must be
// applied.
// In the below steps we approximate the height field profile
// as piecewise linear curve. We find the pair of endpoints
// between which the intersection between the height field
// profile and the view ray is found and then compute line segment
// intersection for the view ray and the line segment formed by
// the two endpoints. This intersection is the displacement
// offset from the original texture coordinate.

float fCurrHeight = 0.0;
float fStepSize   = 1.0 / (float) nNumSteps;
float fPrevHeight = 1.0;
float fNextHeight = 0.0;

int    nStepIndex = 0;
bool   bCondition = true;

float2 vTexOffsetPerStep = fStepSize * i.vParallaxOffsetTS;
float2 vTexCurrentOffset = i.texCoord;
float  fCurrentBound     = 1.0;
float  fParallaxAmount   = 0.0;

float2 pt1 = 0;
float2 pt2 = 0;

float2 texOffset2 = 0;

while ( nStepIndex < nNumSteps )
{
   vTexCurrentOffset -= vTexOffsetPerStep;

   // Sample height map which in this case is stored in the
   // alpha channel of the normal map:
   fCurrHeight = tex2Dgrad( tNormalMap, vTexCurrentOffset,
                            dx, dy ).a;

   fCurrentBound -= fStepSize;

   if ( fCurrHeight > fCurrentBound )
   {
      pt1 = float2( fCurrentBound, fCurrHeight );
      pt2 = float2( fCurrentBound + fStepSize, fPrevHeight );

      texOffset2 = vTexCurrentOffset - vTexOffsetPerStep;

      nStepIndex = nNumSteps + 1;
   }
   else
   {
      nStepIndex++;
      fPrevHeight = fCurrHeight;
   }
}   // End of while ( nStepIndex < nNumSteps )
```

```
     float fDelta2 = pt2.x - pt2.y;
     float fDelta1 = pt1.x - pt1.y;
     fParallaxAmount = (pt1.x * fDelta2 - pt2.x * fDelta1 ) /
                       ( fDelta2 - fDelta1 );
     float2 vParallaxOffset = i.vParallaxOffsetTS *
                       (1 - fParallaxAmount );

     // The computed texture offset for the displaced point
     // on the pseudo-extruded surface:
     float2 texSampleBase = i.texCoord - vParallaxOffset;
     texSample = texSampleBase;

     // Lerp to bump mapping only if we are in between,
     // transition section:
     cLODColoring = float4( 1, 1, 1, 1 );

     if ( fMipLevel > (float)(nLODThreshold - 1) )
     {
        // Lerp based on the fractional part:
        fMipLevelFrac = modf( fMipLevel, fMipLevelInt );

        if ( bVisualizeLOD )
        {
           // For visualizing: lerping from regular POM-
           // resulted color through blue color for transition layer:
           cLODColoring = float4( 1, 1, max(1, 2 * fMipLevelFrac), 1 );
        }

        // Lerp the texture coordinate from parallax occlusion
        // mapped coordinate to bump mapping smoothly based on
        // the current mip level:
        texSample = lerp( texSampleBase, i.texCoord, fMipLevelFrac );

    }  // End of if ( fMipLevel > fThreshold - 1 )

    if ( bDisplayShadows == true )
    {
       float2 vLightRayTS = vLightTS.xy * fHeightMapRange;

       // Compute the soft blurry shadows taking into account
       // self-occlusion for features of the height field:

       float sh0 =  tex2Dgrad( tNormalMap, texSampleBase, dx, dy ).a;
       float shA = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS
           * 0.88, dx, dy ).a - sh0 - 0.88 ) *  1 * fShadowSoftening;
       float sh9 = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS *
            0.77, dx, dy ).a - sh0 - 0.77 ) *  2 * fShadowSoftening;
       float sh8 = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS *
            0.66, dx, dy ).a - sh0 - 0.66 ) *  4 * fShadowSoftening;
       float sh7 = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS *
            0.55, dx, dy ).a - sh0 - 0.55 ) *  6 * fShadowSoftening;
       float sh6 = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS *
            0.44, dx, dy ).a - sh0 - 0.44 ) *  8 * fShadowSoftening;
       float sh5 = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS *
            0.33, dx, dy ).a - sh0 - 0.33 ) * 10 * fShadowSoftening;
       float sh4 = (tex2Dgrad( tNormalMap, texSampleBase + vLightRayTS *
             0.22, dx, dy ).a - sh0 - 0.22 ) * 12 * fShadowSoftening;

        // Compute the actual shadow strength:
        fOcclusionShadow = 1 - max( max( max( max( max( max( shA, sh9 ),
             sh8 ), sh7 ), sh6 ), sh5 ), sh4 );

        // The previous computation overbrightens the image, let's adjust
        // for that:
        fOcclusionShadow = fOcclusionShadow * 0.6 + 0.4;

     }   // End of if ( bAddShadows )

   }   // End of if ( fMipLevel <= (float) nLODThreshold )

   // Compute resulting color for the pixel:
   cResultColor = ComputeIllumination( texSample, vLightTS,
                                       vViewTS, fOcclusionShadow );

   if ( bVisualizeLOD )
   {
      cResultColor *= cLODColoring;
   }

   // Visualize currently computed mip level, tinting the color blue
   // if we are in the region outside of the threshold level:
   if ( bVisualizeMipLevel )
   {
      cResultColor = fMipLevel.xxxx;
   }

   // If using HDR rendering, make sure to tonemap the result color
   // prior to outputting it. But since this example isn't doing that,
   // we just output the computed result color here:
   return cResultColor;


}   // End of float4 ps_main(..)
```

***Listing 2**. Parallax occlusion mapping algorithm implementation. Pixel shader, DirectX 9.0c shader model 3.0*

Chapter 6

Real-Time Atmospheric Effects in Games

Carsten Wenzel[8]
Crytek GmbH

6.1 Motivation

Atmospheric effects, especially for outdoor scenes in games and other interactive applications, have always been subject to coarse approximations due to the computational expense inherent to their mathematical complexity. However, the ever increasing power of GPUs allows more sophisticated models to be implemented and rendered in real-time. This chapter will demonstrate several ways how developers can improve the level of realism and sense of immersion in their games and applications. The work presented here heavily takes advantage of research done by the graphics community in recent years and combines it with novel ideas developed within Crytek to realize implementations that efficiently map onto graphics hardware. In that context, integration issues into game production engines will be part of the discussion.

6.2 Scene depth based rendering

Scene depth based rendering can be described as a hybrid rendering approach borrowing the main idea from deferred shading [Hargreaves04], namely providing access to the depth of each pixel in the scene to be able to recover its original position in world space. This does not imply that deferred shading is a requirement. Rendering in *CryEngine2* for example still works in the traditional sense (i.e. forward shading) yet applies a lot of scene depth based rendering approaches in various scenarios as will be demonstrated in this chapter. What is done instead is decoupling of the actual shading of (opaque) pixels from later application of atmospheric effects, post processing, etc. This allows complex models to be applied while keeping the shading cost relatively moderate as features are implemented in separate shaders. This limits the chances of running into of current hardware shader limits and allows broader use of these effects as they can often be mapped to older hardware as well.

One reoccurring problem in the implementation of scene depth based rendering effects is handling of alpha transparent objects. Just like in deferred shading, you run into the problem of not actually knowing what a pixel’s color / depth really is since generally only one color / depth pair is stored but pixel overdraw is usually greater

[8] carsten@crytek.de

than one and potentially unbounded (pathologic case but troublesome for hardware designers). Essentially it comes down to the problem of order independent transparency (OIT) for which to date no solution exists on consumer hardware. Possible approaches like A-Buffers are very memory intensive and not programmable. At this point, it is up to the developer to work around the absence of OIT depending on the effect to be implemented (see below).

To make per-pixel depth available for rendering purposes, there are several options. Ideally, depth is laid out in an early Z-Pass (encouraged by the IHVs in order to improve efficiency of early Z-Culling which can tremendously cut down subsequent pixel shading cost) filling the Z-Buffer. Since scene depth based rendering approaches don’t modify scene depth, the Z-Buffer could be bound as a texture to gain access to a pixel’s depth (though this would require a remapping of the Z-Buffer value fetched from post perspective space into eye space) or more conveniently it could be an input to a pixel shader automatically provided by the GPU. Due to limitations in current APIs, GPUs and shading models, this is unfortunately not possible for the sake of compatibility but should be a viable, memory saving option in the near future. For the time being, it is necessary to explicitly store scene depth in an additional texture. In *CryEngine2* this depth texture is stored as linear, normalized depth (0 at camera, 1 at far clipping plane) which will be important later when recovering the pixel’s world space position. The format can be either floating point or packed RGBA8. On DirectX9 with no native packing instructions defined in the shader model, the use of RGBA8 although precision-wise comparable to floating point, is inferior in terms of rendering speed due to the cost of encoding / decoding depth. However, it might be an option on OpenGL where vendor specific packing instructions are available. An issue arises in conjunction with multisample anti-aliasing (MSAA). In this case, laying out depth requires rendering to a multi-sampled buffer. To be able to bind this as a texture, it needs to be resolved. Currently there’s no way to control that down-sampling process. As a result depth values of object silhouettes will merge with the background producing incorrect depth values which will later be noticeable as seams.

Given the depth of a pixel, recovering its world space position is quite simple. A lot of deferred shading implementations transform the pixel’s homogenous coordinates from post perspective space back into world space. This takes three instructions (three `dp4`’s or `mul` / `mad` assembly instructions) since we don’t care about w (it’s 1 anyway). However there’s often a simpler way, especially when implementing effects via full screen quads. Knowing the camera’s four corner points at the far clipping plane in world space, a full screen quad is set up using the distance of each of these points from the camera position as input texture coordinates. During rasterization, this texture coordinate contains the current direction vector from the camera to the far clipping plane. Scaling this vector by the linear, normalized pixel depth and adding the camera position yields the pixel’s position in world space. This only takes one mad instruction and the direction vector comes in for free thanks to the rasterizer.

A lot of the techniques described in this chapter exploit the availability of scene depth at any stage after the actual scene geometry has been rendered to map pixels back to their source location in world space for various purposes.

6.3 Sky light rendering

Probably the most fundamental part of rendering outdoor scenes is a believable sky that changes over time. Several methods with varying quality and complexity have been developed over the years. To accurately render sky light in *CryEngine2*, the model proposed in [Nishita93] was implemented as it enables rendering of great looking sunsets and provides several means for artist controllability over the output. Unfortunately, it is also one of the most computationally expensive models around. [O'Neil05] presents an implementation for the general case (flight simulator) which runs entirely on the GPU which required several simplifications and tradeoffs to make it work. Using these, it was possible to squeeze Nishita's model into the limits of current hardware but it came at the price of rendering artifacts (color gradients occasionally showed "layer-like" discontinuities).

The goal for *CryEngine2* was to get the best quality possible at reasonable runtime cost by trading in flexibility in camera movement using the following assumption. The viewer is always on the ground (zero height) which is fairly reasonable for any type of game where it's not needed to reach into upper atmosphere regions. This means the sky only ever needs to update when time changes and the update is completely independent of camera movement. The implementation used the acceleration structures suggested by the original paper. That is, we assume sun light comes in parallel and can hence build a 2D lookup table storing the angle between incoming sunlight and the zenith and the optical depth for the height of a given atmosphere layer (exponentially distributed according to characteristics of atmosphere). A mixed CPU / GPU rendering approach was chosen since solving the scattering integral involves executing a loop to compute intermediate scattering results for the intersections of the view ray with *n* atmosphere layers for each point sampled on the sky hemisphere. On the CPU, we solve the scattering integral for 128x64 sample points on the sky hemisphere using the current time of day, sunlight direction as well as Mie and Rayleigh scattering coefficients and store the result in a floating point texture. A full update of that texture is distributed over several fames to prevent any runtime impact on the engine. One distributed update usually takes 15 to 20 seconds. On CPU architectures providing a vectorized expression in their instruction set (e.g. consoles such as Xbox360 and PS3) the computation cost can be significantly reduced. This texture is used each frame by the GPU to render the sky. However, since the texture resolution would result in very blocky images, a bit of work is offloaded to the pixel shader to improve quality. By computing the phase function on the fly per pixel (i.e. not pre-baking it into the low resolution texture) and multiplying it to the scattering result for a given sample point on the sky hemisphere (a filtered lookup from the texture) it is possible to remove the blocky artifacts completely even around the sun where luminance for adjacent pixel varies very rapidly.

On GPUs with efficient dynamic branching it might be possible to move the current approach to solving the scattering integral completely over to GPU. Thanks to the 2D lookup table, the code is already quite compact. Initial test of porting the C++ version of the solver over to HLSL showed that, using loops, it would translate to approx. 200 shader instructions. Currently the loop executes $n = 32$ times. This number of exponentially distributed atmosphere layers was found sufficient to produces precise enough integration results to yield a good looking sky even for "stress tests" like rendering a sunset. Considering that approx. $32 * 200 = 6400$ instructions would have to be executed to solve the scattering integral for each sample point on the sky hemisphere, it seems necessary to distribute the update over several frames (e.g. consecutively rendering part of a subdivided quad into the texture updating individual parts). But still the update rate should be significantly shorter than it is right now.

6.4 Global volumetric fog

Even though Nishita's model indisputably produces nice results, it is still way too expensive to be applied to everything in a scene (i.e. to compute in and out scattering along view ray to the point in world space representing the current pixel in order to model aerial perspective).

Still, it was desired to provide an atmosphere model that can apply its effects on arbitrary objects with the scene. This section will first propose the solution implemented in *CryEngine2*. How to make it interact with the way sky light is computed will be described in the next section. It follows the derivation of an inexpensive formula to compute height/distance based fog with exponential falloff.

$$
\begin{aligned}
&f\left((x,y,z)^T\right)=be^{-cz}\\
&\vec{v}(t)=\vec{o}+t\vec{d}\\
&\oint f(\vec{v}(t))dt=\int_0^1 f\left((o_x+td_x,o_y+td_y,o_z+td_z)^T\right)\left\|\vec{d}\right\|dt\\
&=\int_0^1 be^{-co_z-ctd_z}\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}dt\\
&=be^{-co_z}\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\int_0^1 e^{-ctd_z}dt\\
&=be^{-co_z}\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\left[-\frac{e^{-ctd_z}}{cd_z}\right]_0^1\\
&=be^{-co_z}\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\left[-\frac{e^{-cd_z}}{cd_z}+\frac{1}{cd_z}\right]\\
&=be^{-co_z}\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\left[\frac{1-e^{-cd_z}}{cd_z}\right]\\
&F(\vec{v}(t))=e^{-\oint f(\vec{v}(t))dt}
\end{aligned}
$$

f - fog density distribution function
b - global density
c - height falloff
v - view ray from camera (o) to world space pos of pixel (o+d), t=1
F - fog density along v

This translates to the following piece of HLSL code (Listing 1). Care must be taken in case the view ray looks precisely horizontal into the world (as d_z is zero in that case).

```
float ComputeVolumetricFog( in float3 worldPos, in float3
cameraToWorldPos)
{
  // NOTE:
  // cVolFogHeightDensityAtViewer = exp( -cHeightFalloff *
  //                                     vfViewPos.z );
  float fogInt = length( cameraToWorldPos ) *
                 cVolFogHeightDensityAtViewer;

  const float cSlopeThreshold = 0.01;
  if( abs( cameraToWorldPos.z ) > cSlopeThreshold )
  {
     float t = cHeightFalloff * cameraToWorldPos.z;
     fogInt *= ( 1.0 - exp( -t ) ) / t;
  }

  return exp( -cGlobalDensity * fogInt );
}
```

Listing 1. *Computing volumetric fog in a DirectX 9.0c HLSL shader code block*

The “`if`” condition which translates into a `cmp` instruction after compiling the shader code prevents floating point specials from being propagated which would otherwise wreak havoc at later stages (tone mapping, post processing, etc). The code translates into 18 instructions using the current version of the HLSL compiler and shader model 2.0 as the target.

Calling this function returns a value between zero and one which can be used to blend in fog. For all opaque scene geometry, this model can be applied after the opaque geometry rendering pass by simply drawing a screen size quad setting up texture coordinates as described in Section 7.2 and invoking that function per pixel. What remains to be seen is how to calculate a fog color that is a good match to blend with the sky. This will be topic of the next section.

6.5 Combining sky light and global volumetric fog

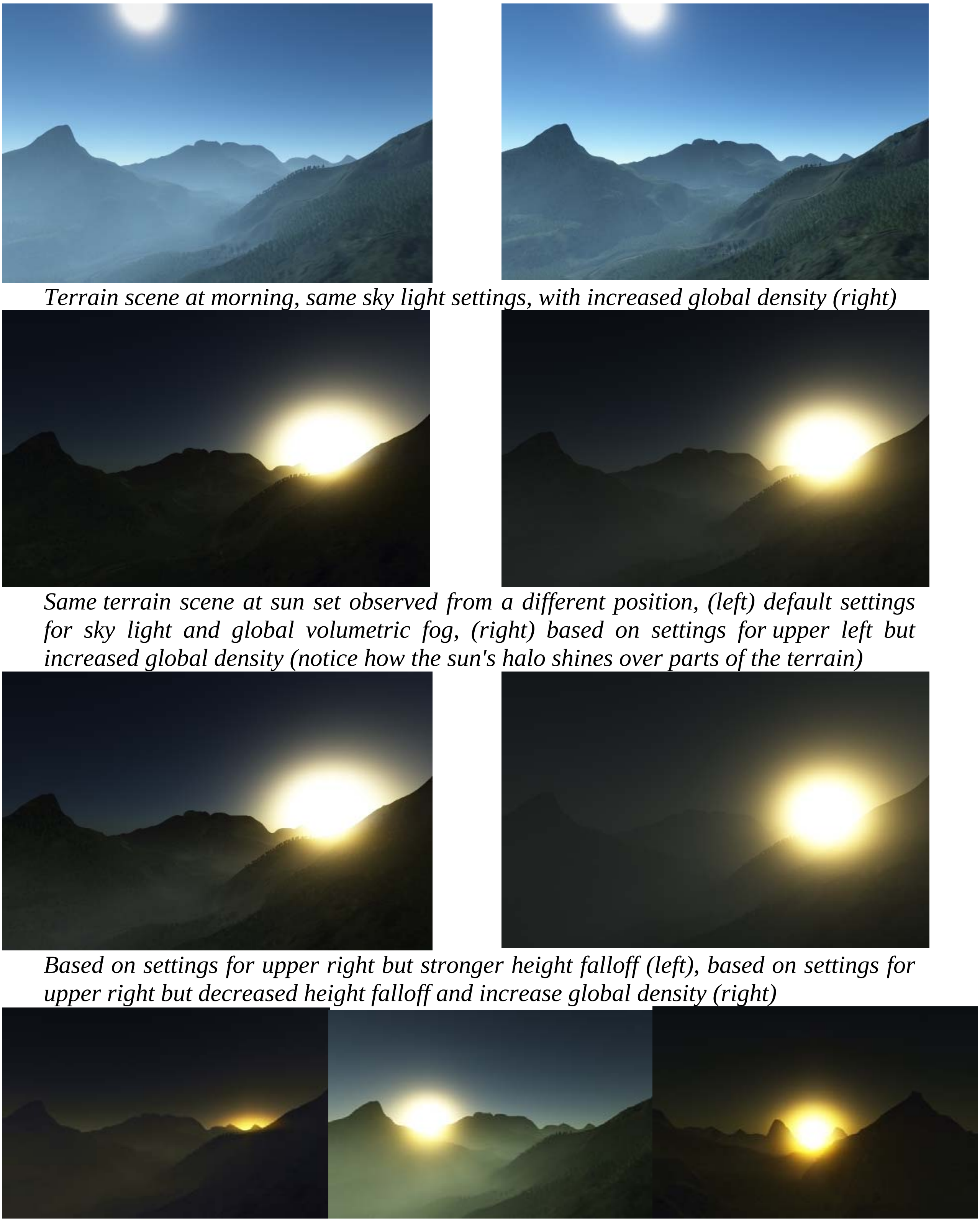
We create a slight problem by implementing a separate model for sky light and global volumetric fog. Now we have two models partially solving the same problem: How to render atmospheric fog / haze. The question is whether it is possible for these two models be combined to work together. We certainly wish to achieve halos around the sun when setting up hazy atmosphere conditions, realize nice color gradients to get a feeling of depth in the scene, be able to see aerial perspective (i.e. the color gradient of a mountain in the distance which is partiality in fog should automatically correlate with the colors of the sky for a given time and atmospheric settings). Nishita’s model would allow rendering that but is too expensive to be used in the general case. The global volumetric fog model presented in the previous section is suitable for real-time rendering but far more restrictive.

To make the two models cooperate, we need a way to determine a fog color that can be used with accompanied volumetric fog value to blend scene geometry nicely into the sky. For that purpose, Nishita’s model was enhanced slightly to allow a low cost per-pixel computation of a fog color matching the sky’s color at the horizon for a given direction. To do this, all samples taken during the sky texture update for directions resembling the horizon are additionally averaged and stored for later use in a pixel shader (using all of the horizon samples to produce a better matching fog color seems tempting but it was found that the difference to the average of all horizon samples is barely noticeable and doesn't justify the additional overhead in shading computations). When rendering the fog, the same code that calculated the final sky color can be used to gain a per-pixel fog color. The phase function result is computed as before but instead of accessing the low resolution texture containing the scattering results, we use the average of the horizon samples calculated on the CPU. Now using the volumetric fog value computed for a given pixel the color stored in the frame buffer can be blended against the fog color just determined. This may not be physically correct but gives pleasing results at very little computational cost.

6.6 Locally refined fog via volumes

For game design purposes, it is often necessary to locally hide or disguise certain areas in the game world and global fog is not really suited for that kind of purpose. Instead fog volumes come into play. The implementation goal was to be able to apply the same kind of fog model as described in Section 7.4 to a locally refined area. The model was slightly enhanced to allow the fog gradient to be arbitrarily oriented. We support two types of fog volume primitives, namely ellipsoids and boxes.

As can be seen in the derivation of the formula for global volumetric fog, a start and end point within the fog volume are needed to solve the fog integral. (Note that the global fog can be thought of as an infinite volume where the start point represents the camera position and end point is equal to the world space position of current pixel.) To determine these two points, we actually render the bounding hull geometry of the fog volume, i.e. a box whose front faces are rendered with Z-Test enabled as long as the camera is outside the fog volume and whose back faces are rendered with Z-Test disabled once the camera is inside the volume. This way it is possible to trace a ray for each pixel to find out if and where the view ray intersects the fog volume. Ray tracing for both primitive types happens in object space for simplicity and efficiency (i.e. transforming the view ray into object space and checking against either a unit sphere or unit cube and transforming the results back into world space). If no intersection occurs, that pixel is discarded via HLSL's `clip` instruction. Doing that has the additional side effect of simplifying the code for shader models and/or hardware not supporting (efficient) branching (i.e. it compiles to less instructions) since it avoids the need to consider "`if / else`" cases for which otherwise all branches would have to be executed and a final result picked. If an intersection occurs, an additional check using the pixel's depth value is necessary to determine if scene geometry is hit before exiting the fog volume. In that case this point overrides the end point. Also, in case we're inside the volume, we need to ensure that the start point is always the camera point. With the start and end point known, these can be plugged into the volumetric fog formula earlier to compute a fog value per pixel.



Terrain scene at morning, same sky light settings, with increased global density (right)

Same terrain scene at sun set observed from a different position, (left) default settings for sky light and global volumetric fog, (right) based on settings for upper left but increased global density (notice how the sun's halo shines over parts of the terrain)

Based on settings for upper right but stronger height falloff (left), based on settings for upper right but decreased height falloff and increase global density (right)

Various sunset shots with different setting for global density, height falloff, Mie- and Rayleigh scattering

Figure 1. *Atmospheric scattering effects.*

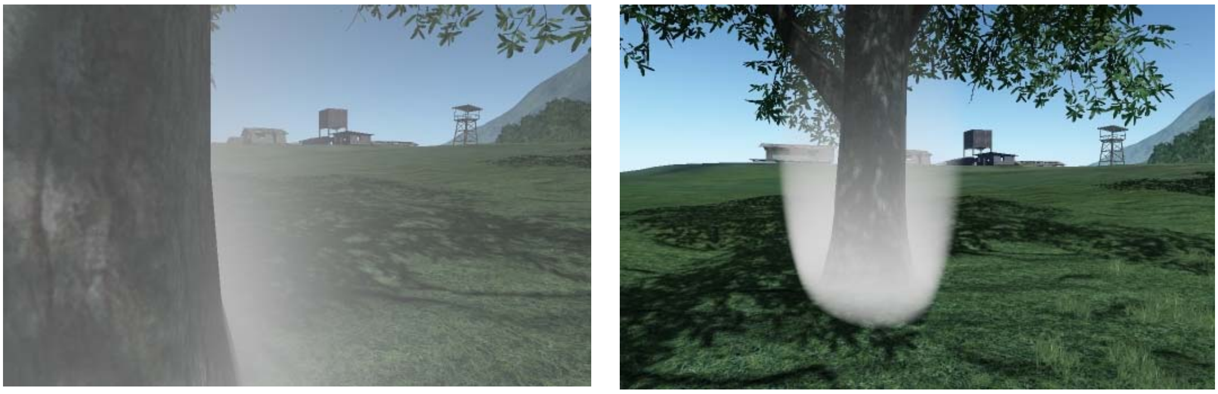


***Figure 2**. GPU ray traced fog volume (ellipsoid), observed from inside (left) and outside (right)*

Other approaches to implementing fog volumes were also considered. Polygonal fog volumes seemed like a good idea at first (depth of all back faces is accumulated and subtracted from depth of all front faces to get length traveled through volume along pixel direction). They also don't suffer from clipping artifacts at the near clip plane since a depth of zero doesn't have any impact while accumulating front / back face depth. Their advantage over the fog volume primitives described above is that they can be arbitrarily complex and convex. In order to do efficient ray tracing one is currently still stuck with rather simplistic primitives like ellipsoids and boxes. However, polygonal fog volumes also exhibit a few disadvantages which have to be taken into account and outweigh their advantages (at least for the purpose of *CryEngine2*). First they really allow depth based fog only. Currently they need to be rendered in two passes to accumulate back face depth and subtract front face depth. Moreover to do so they need additional texture storage and floating point blending support which is kind of prohibitive when fog volume support is required for older hardware as well. Implementations of polygonal fog volumes exist that use clever bit-twiddling to make them work with standard RGBA8 textures but then it is necessary volumes do not exceed a certain depth complexity or size to prevent overflow which creates another limitation and kind of defeats their usefulness in the first place.

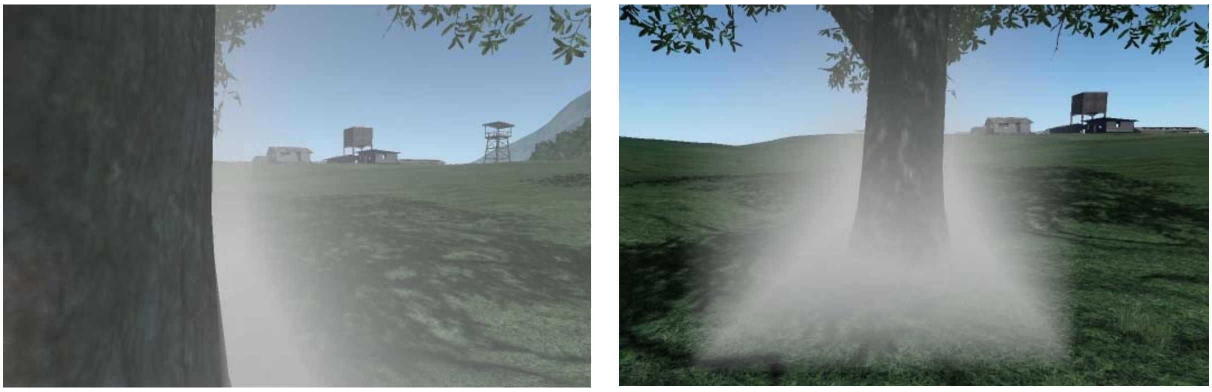


***Figure 3**. GPU ray traced fog volume (box), observed from inside (left) and outside (right)*

6.7 Fogging alpha transparent objects

As mentioned in the introduction, scene depth based rendering approaches often cause problems with alpha transparent objects, in this case the global/local volumetric fog model. Since currently there's no feasible hardware solution available to tackle this problem, it is up to the developer to find suitable workarounds, as will be shown in this section.

6.7.1 Global fog

Global volumetric fog for alpha transparent objects is computed per vertex. Care needs to be taken to properly blend the fogged transparent object into the frame buffer already containing the fogged opaque scene. It proves to be very useful that the entire fog computation (fog density as well as fog color) is entirely based on math instructions and doesn't require lookup tables of any sort. The use of textures would be prohibitive as lower shader models don't allow texture lookups in the vertex shader but need to be supported for compatibility reasons.

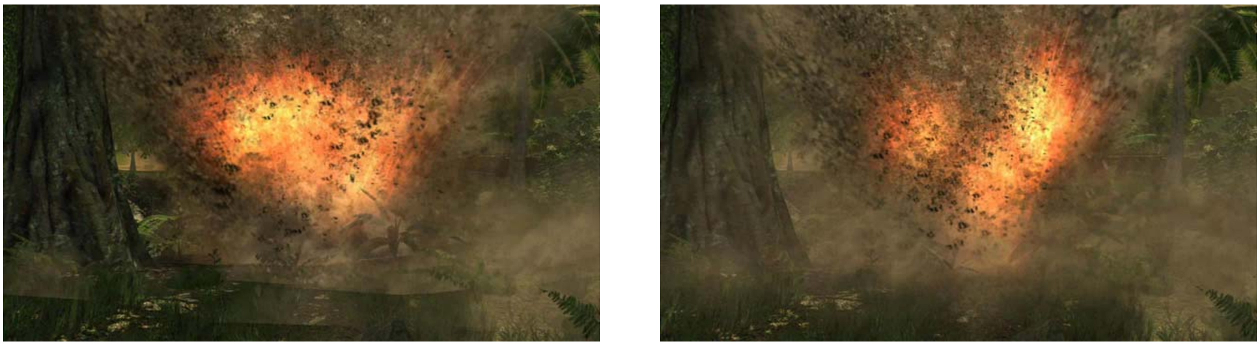
6.7.2 Fog volumes

Applying fog volumes on alpha transparent objects is more complicated. Currently, the contribution of fog volumes on alpha transparent objects is computed per object. This appears to be the worst approximation of all but it was found that if designers know about the constraints implied they can work around it. Usually they need to make sure alpha transparent objects don't become too big. Also, the gradient of fog volumes should be rather soft as sharp gradients make the aliasing problem more obvious (i.e. one sample per object). The ray tracing code done per pixel on the GPU can be easily translated back to C++ and invoked for each alpha transparent objected pushed into the pipeline. A hierarchical structure storing the fog volumes is beneficial to reduce the number of ray/volume traces as much as possible. To compute the overall contribution of all fog volumes (partially) in front of an alpha transparent object the ray tracing results are weighted to a single contribution value in back to front order (i.e. farthest fog volumes gets weighted in first). Another approach that was investigated involved building up a volume texture containing fog volume contributions for sample points around or in front of the camera (i.e. world space or camera space respectively). Both a uniform and non-uniform distribution of sampling points was tried but aliasing was just too bad to deem this approach useful.

One potential extension to the computation of the overall contribution of fog volumes on alpha transparent objects is to compute values for all corners of an object's bounding box and in the vertex shader lerp between them based on the vertex' relative position.

6.8 Soft particles

Particles are commonly used to render various natural phenomena like fire, smoke, clouds, etc. Unfortunately, at least on consumer hardware available today (even with MSAA enabled), they usually suffer from clipping artifacts at their intersection with opaque scene geometry. By having the depth of all opaque objects in the world laid out and accessible in the pixel shader, it is possible to tweak a particle's alpha value per pixel to remove jaggy artifacts as shown in the figure below.



Particles drawn as hard billboards *Soft particles*

Figure 4. *Billboard and soft particle rendering comparison*

Each particle is treated as a screen aligned volume with a certain size. For each pixel, the particle shader determines how much the view ray travels through the particle volume until it hits opaque scene geometry. Dividing that value by the particle volume size and clamping the result to [0, 1] yields a relative percentage of how much opaque scene geometry penetrates the particle volume for the given pixel. It can be multiplied with the original alpha value computed for the current particle pixel to softly fade out the particle wherever it's getting close to touching opaque geometry.

6.9 Other effects benefiting from per pixel depth access

6.9.1 Clouds

The cloud rendering subsystem in *CryEngine2* is based on [Wang03]. Shading is gradient-based and scene depth used to implement soft clipping, especially with terrain (e.g. rain clouds around mountains) and both the near and far clipping plane. It borrows the main idea from the soft particle implementation. Additionally back lighting with respect to the sun was added to achieve the effect of glowing edges when looking at clouds partially covering the sun.

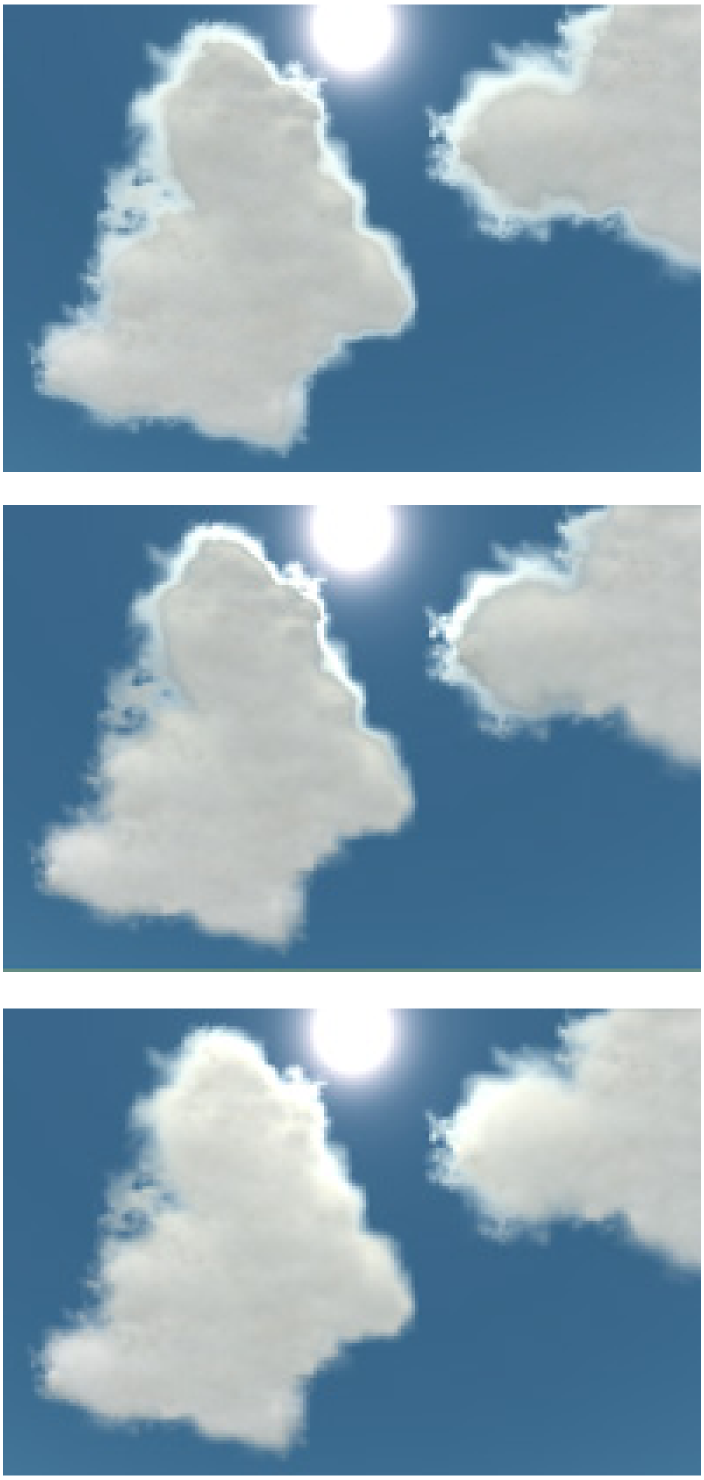
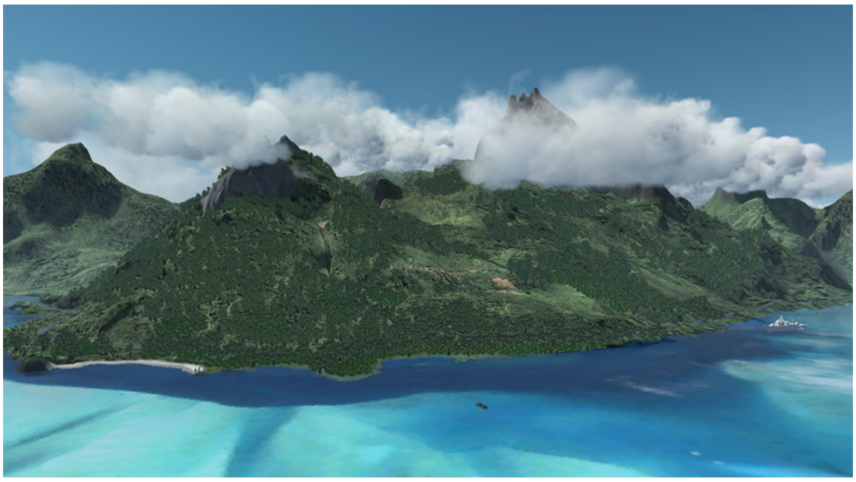
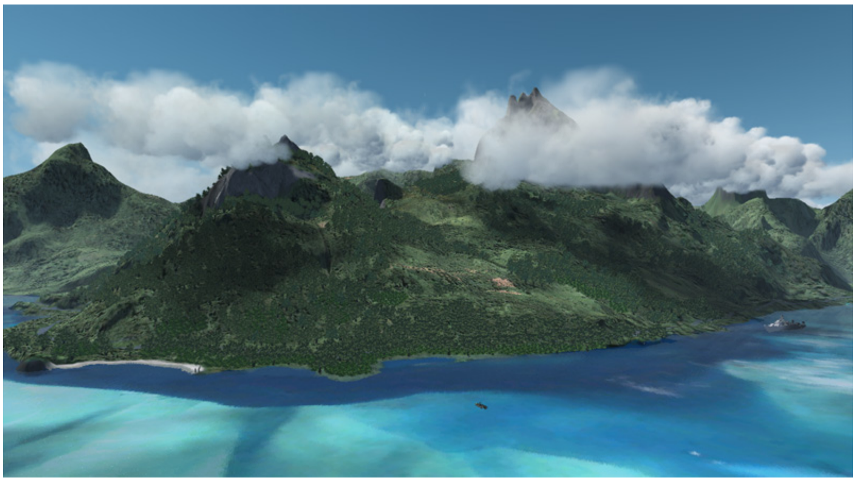


***Figure 5**. Backlighting of clouds, showing different values for backlighting threshold (with respect to cloud alpha), and backlighting softness*

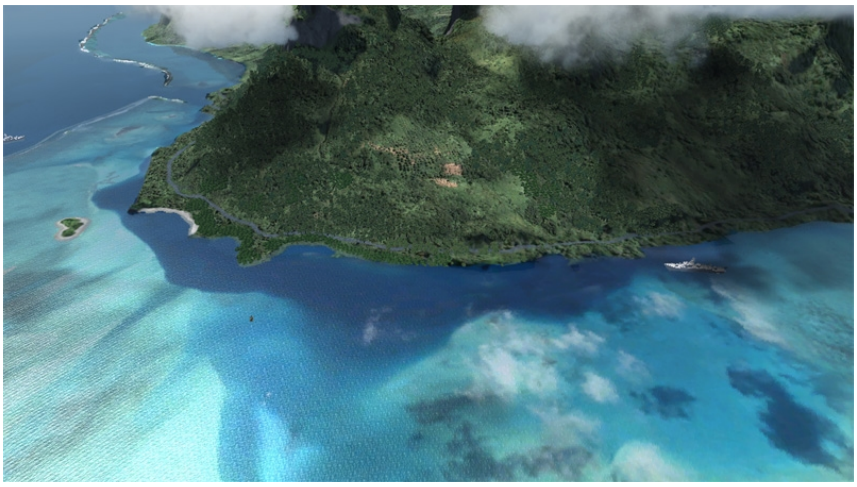
Cloud shadows are cast in single full screen pass using the scene depth to recover world space position to be able to transform into shadow map space.



Island scene without cloud shadows



Same scene with cloud shadows. Notice how it breaks up the regularity of shading giving a more natural look



Same scene with cloud shadows viewed from a different position

***Figure 6**. Cloud shadow rendering*

6.9.2 Volumetric lightning

Modeling volumetric lightning is similar to the global volumetric fog model postulated earlier. Only this time it is light emitted from a point falling off radially. The attenuation function needs to be chosen carefully in order to be able to integrate it in a closed form.

$$f\left((x,y,z)^T\right)=\frac{i}{1+a\cdot\left\|\vec{l}-(x,y,z)^T\right\|^2}$$

$$\vec{v}(t)=\vec{o}+t\vec{d}$$

$$\oint f\left(\vec{v}(t)\right)dt=\int_0^1 f\left((o_x+td_x,o_y+td_y,o_z+td_z)^T\right)\left\|\vec{d}\right\|dt$$

$$=\int_0^1\left(\frac{i}{1+a\left\|\vec{l}-\left(\vec{o}+t\vec{d}\right)\right\|^2}\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\right)dt$$

$$\left\langle\vec{c}=\vec{l}-\vec{o}\right\rangle$$

$$=i\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\int_0^1\left(\frac{1}{1+a\cdot\left\|\vec{c}-t\cdot\vec{d}\right\|^2}\right)dt$$

$$=i\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\int_0^1\left(\frac{1}{1+a\left((c_x-td_x)^2+\left(c_y-td_y\right)^2+\left(c_y-td_y\right)^2\right)}\right)dt$$

$$\left\langle u=1+a(\vec{c}\circ\vec{c});v=-2a\left(\vec{c}\circ\vec{d}\right);w=a\left(\vec{d}\circ\vec{d}\right)\right\rangle$$

$$=i\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\int_0^1\left(\frac{1}{u+vt+wt^2}\right)dt$$

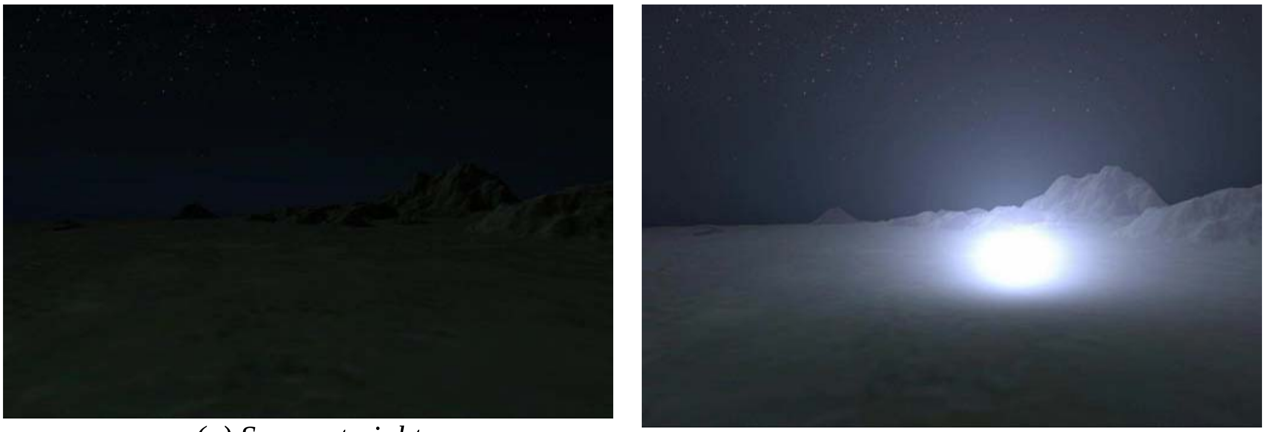
$$=i\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\left[\frac{2\arctan\left(\frac{v+2wt}{\sqrt{4uw-v^2}}\right)}{\sqrt{4uw-v^2}}\right]_0^1$$

$$=2i\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\left[\frac{\arctan\left(\frac{v+2w}{\sqrt{4uw-v^2}}\right)-\arctan\left(\frac{v}{\sqrt{4uw-v^2}}\right)}{\sqrt{4uw-v^2}}\right]$$

$$=F\left(\vec{v}(t)\right)$$

f - light attenuation function
i - source light intensity
a - global attenuation control value
v - view ray from camera (o) to world space pos of pixel (o+d), t=1
F - amount of light gathered along v

This lightning model can be applied just like global volumetric fog by rendering a full screen pass. By tweaking the controlling variables a and i volumetric lightning flashes can be modeled. F represents the amount of light emitted from the lightning source that gets scattered into the view ray. Additionally, when rendering scene geometry the lightning source needs to be taken into account when computing shading results.



(a) Scene at night

(b) Same scene rendering as the lightning strikes

***Figure 7**. Volumetric lightning rendering.*

6.10 Conclusion

GPUs nowadays offer a lot of possibilities to realize complex visual effects in real-time. This chapter has shown a few examples of atmospheric effects implemented within *CryEngine2,* how scene depth based rendering was utilized in them, what integration issues had to be faced and how they got solved.

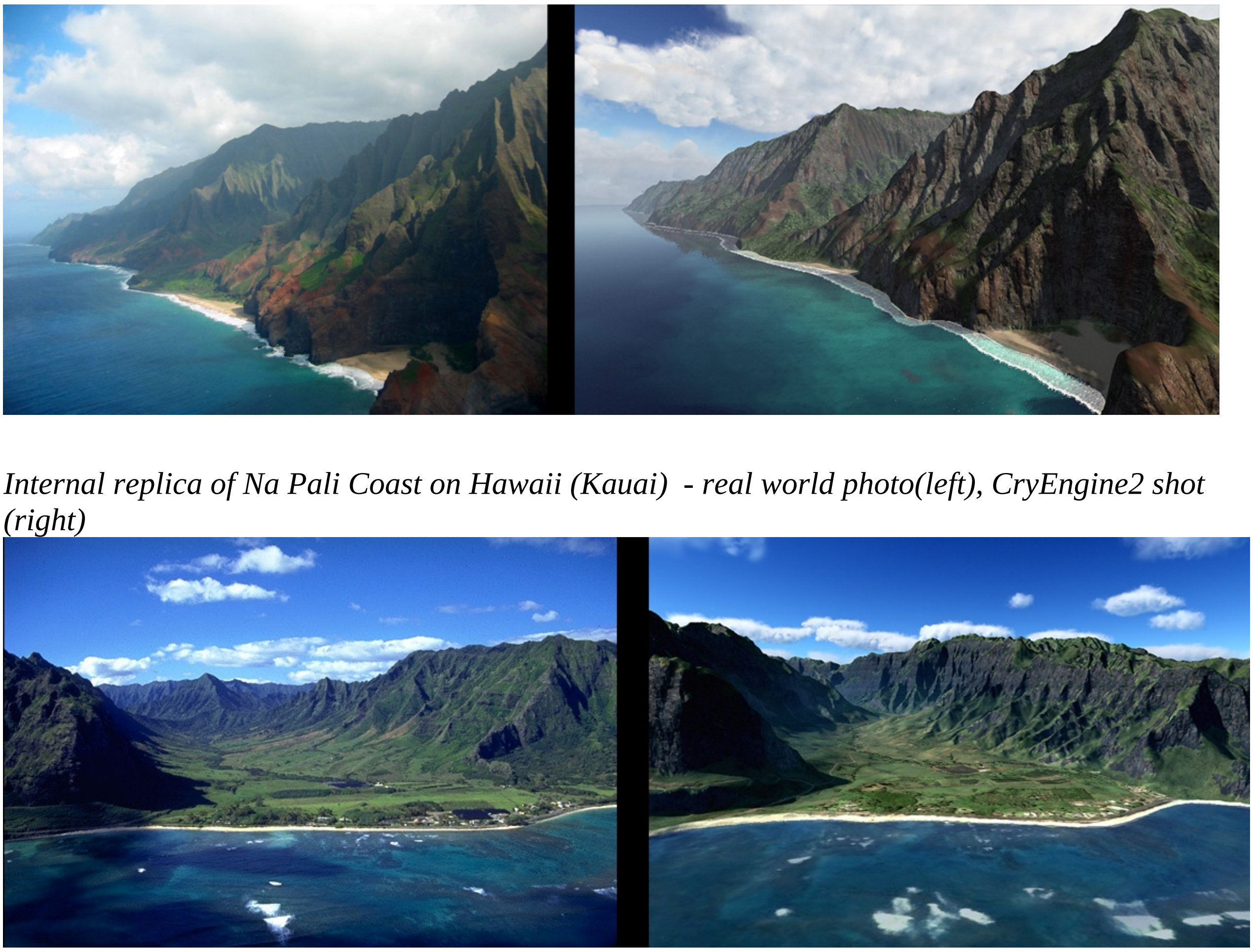
6.11 Bibliography

Hargreaves, S. 2004. Deferred Shading, *Game Developers Conference*, D3D Tutorial Day, March, 2004.

Nishita, T., Sirai, T., Tadamura, K., Nakamae, E. 1993. Display of the Earth Taking into Account Atmospheric Scattering, Proceedings of the 20th annual conference on Computer graphics and interactive techniques *(SIGGRAPH '93)*, pp. 175-182.

O'Neal, S. 2005. Accurate Atmospheric Scattering. GPU Gems 2, Addison Wesley, pages 253-268

Wang, N. 2003. Realistic and Fast Cloud Rendering in Computer Games. Proceedings of the SIGGRAPH 2003 conference. Technical Sketch.



Internal replica of Na Pali Coast on Hawaii (Kauai) - real world photo(left), CryEngine2 shot (right)

Internal replica of Kualoa Ranch on Hawaii - real world photo(left), CryEngine2 shot (right)

Figure 8. *In-game rendering comparisons with the real-world photographs.*

Chapter 7

Shading in Valve's Source Engine

Jason Mitchell[9], Gary McTaggart[10] and Chris Green[11]

VALVE

7.1 Introduction

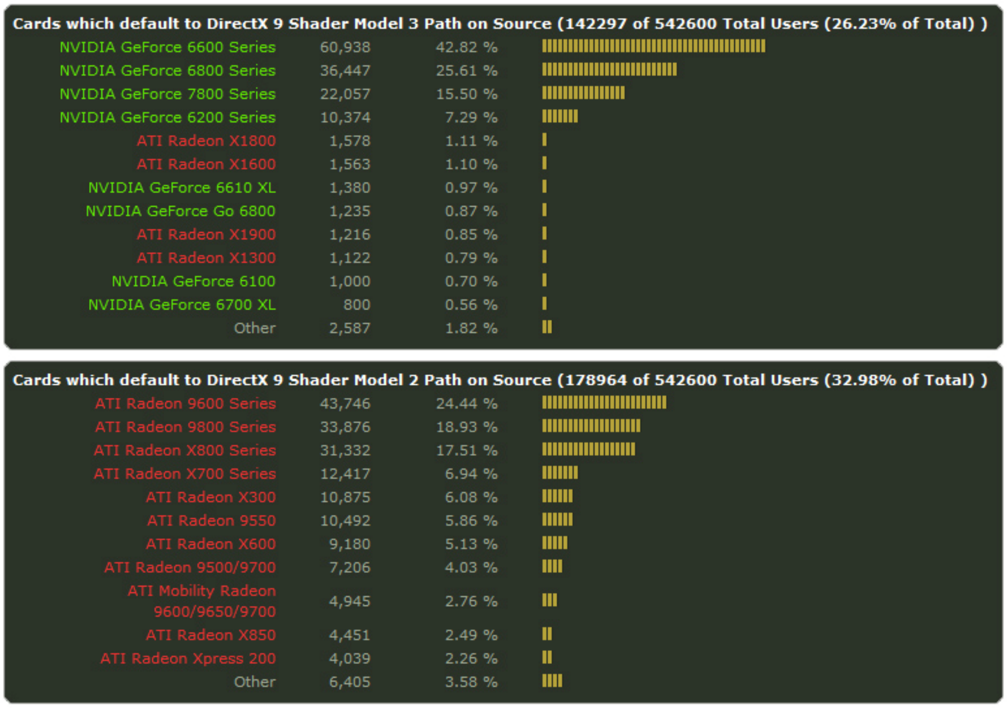
Starting with the release of *Half-Life 2* in November 2004, Valve has been shipping games based upon its Source game engine. Other Valve titles using this engine include *Counter-Strike: Source*, *Lost Coast*, *Day of Defeat: Source* and the recent *Half-Life 2: Episode 1*. At the time that *Half-Life 2* shipped, the key innovation of the Source engine's rendering system was a novel world lighting system called *Radiosity Normal Mapping*. This technique uses a novel basis to economically combine the soft realistic

[9] jasonm@valvesoftware.com
[10] gary@valvesoftware.com
[11] cgreen@valvesoftware.com

lighting of radiosity with the reusable high frequency detail provided by normal mapping. In order for our characters to integrate naturally with our radiosity normal mapped scenes, we used an *irradiance volume* to provide directional ambient illumination in addition to a small number of local lights for our characters. With Valve's recent shift to episodic content development, we have focused on incremental technology updates to the Source engine. For example, in the fall of 2005, we shipped an additional free *Half-Life 2* game level called *Lost Coast* and the multiplayer game *Day of Defeat: Source*. Both of these titles featured real-time High Dynamic Range (HDR) rendering and the latter also showcased the addition of real-time c*olor correction* to the engine. In these course notes, we will describe the unique aspects Valve's shading techniques in detail.

7.2 Engineering / Marketplace Context

Before describing the details of the Source engine's shading techniques, it is important to note that the decision to include particular features must be made in the context of the marketplace. In order to make informed technology tradeoffs, we periodically perform a voluntary survey of approximately one million of our users' PCs via our online distribution system, *Steam*. This survey, which is publicly available at http://www.steampowered.com/status/survey.html, can tell us, for example, how many of our users have video cards capable of running ps_2_b pixel shaders (40%) or how many of our users have 256MB or more video memory (38%). Having an accurate picture of our users' hardware helps us to weigh the engineering cost of adopting a given graphics feature against the benefit to our end users. All of the decisions that drive development in the Source engine are based upon these kinds of measured tradeoffs.



Cards which default to DirectX 9 Shader Model 3 Path on Source (142297 of 542600 Total Users (26.23% of Total))

NVIDIA GeForce 6600 Series	60,938	42.82 %
NVIDIA GeForce 6800 Series	36,447	25.61 %
NVIDIA GeForce 7800 Series	22,057	15.50 %
NVIDIA GeForce 6200 Series	10,374	7.29 %
ATI Radeon X1800	1,578	1.11 %
ATI Radeon X1600	1,563	1.10 %
NVIDIA GeForce 6610 XL	1,380	0.97 %
NVIDIA GeForce Go 6800	1,235	0.87 %
ATI Radeon X1900	1,216	0.85 %
ATI Radeon X1300	1,122	0.79 %
NVIDIA GeForce 6100	1,000	0.70 %
NVIDIA GeForce 6700 XL	800	0.56 %
Other	2,587	1.82 %

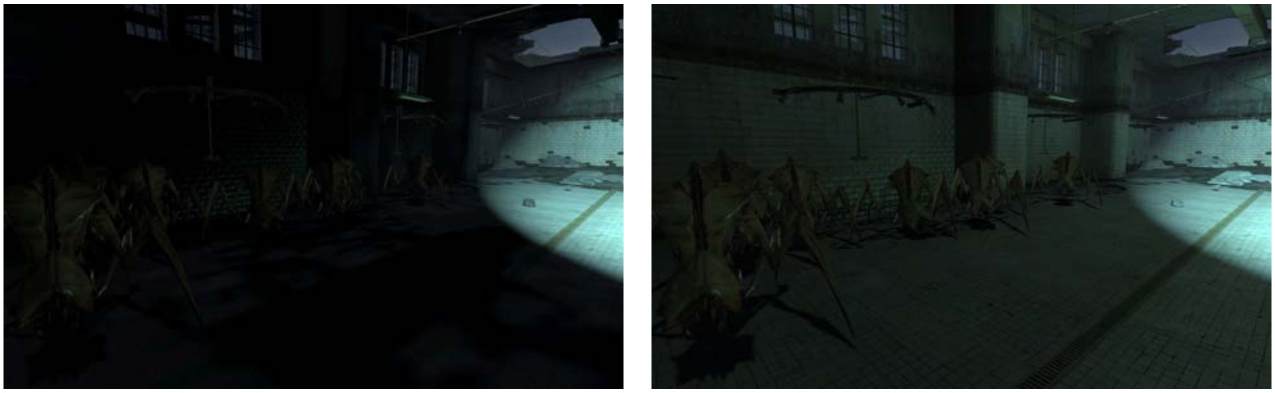
Cards which default to DirectX 9 Shader Model 2 Path on Source (178964 of 542600 Total Users (32.98% of Total))

ATI Radeon 9600 Series	43,746	24.44 %
ATI Radeon 9800 Series	33,876	18.93 %
ATI Radeon X800 Series	31,332	17.51 %
ATI Radeon X700 Series	12,417	6.94 %
ATI Radeon X300	10,875	6.08 %
ATI Radeon 9550	10,492	5.86 %
ATI Radeon X600	9,180	5.13 %
ATI Radeon 9500/9700	7,206	4.03 %
ATI Mobility Radeon 9600/9650/9700	4,945	2.76 %
ATI Radeon X850	4,451	2.49 %
ATI Radeon Xpress 200	4,039	2.26 %
Other	6,405	3.58 %

***Figure 1.** Typical Survey Measurements - sampled from 1 million gamers in Spring 2006*

7.3 World Lighting

For *Half-Life 2*, we leveraged the power of light maps generated by our in-house radiosity solver, *vrad*, while adding high frequency detail using per-pixel normal information. On its own, radiosity has a lot of nice properties for interactive content in that it is possible to pre-generate relatively low resolution light maps which result in convincing global illumination effects at a low run-time cost. A global illumination solution like this requires less micro-management of light sources during content production and often avoids objectionable harsh lighting situations as shown in the images below.



Harsh Direct Lighting Only *Radiosity*

***Figure 2**. Benefits of Radiosity Lighting*

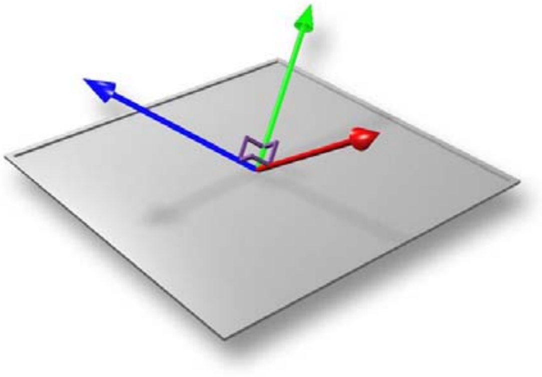
Because of the unpredictable interactive nature of our medium—we don’t know where the camera will be, since the player is in control—we can’t tune lights shot-by-shot as in film production. Global illumination tends to result in game environments that look pleasing under a variety of viewing conditions. Because of this, heavily favoring the look of global illumination is a common theme in Valve’s shading philosophy, as we will discuss later in the section on model lighting via an irradiance volume.

7.3.1 Radiosity Normal Mapping

While radiosity has a lot of strengths, the resulting low-resolution light maps can’t capture the high-frequency detail we would like to author into our surfaces. Even at relatively modest luxel density—a luxel is typically 4” by 4” in our game worlds—light maps require a significant amount of video memory in a typical game level since they are unique to a particular polygon or location in the world. While many current games accumulate one local light per rendering pass [Diefenbach97] or perform several local lighting computations in a single pixel shader [Franke06], we seek to effectively perform diffuse bump mapping with respect to an arbitrary number of lights in one pass.

With high frequency surface detail in the form of repeatable/reusable albedo and normal maps that are higher resolution than our 4" by 4" per luxel light maps, we combined the strengths of normal mapping and light mapping in a novel technique we call *Radiosity Normal Mapping*. Radiosity Normal Mapping is the key to the Source engine's world lighting algorithm and overall look [McTaggart04].

The core idea of Radiosity Normal Mapping is the encoding of light maps in a novel basis which allows us to express directionality of incoming radiance, not just the total cosine weighted incident radiance, which is where most light mapping techniques stop. Our offline radiosity solver has been modified to operate on light maps which encode directionality using the basis shown in the figure below.



$$\left\{-\frac{1}{\sqrt{6}},\frac{1}{\sqrt{2}},\frac{1}{\sqrt{3}}\right\}$$

$$\left\{-\frac{1}{\sqrt{6}},-\frac{1}{\sqrt{2}},\frac{1}{\sqrt{3}}\right\}$$

$$\left\{\sqrt{\frac{2}{3}},0,\frac{1}{\sqrt{3}}\right\}$$

***Figure 3**. Radiosity Normal Mapping Basis*

That is, at the cost of tripling the memory footprint of our light maps, we have been able to encode directionality in a way that is compatible with conventional Cartesian normal maps stored in surface local coordinates. Because of this encoding, we are able to compute exit radiance as a function of albedo, normal and our directional light maps so that we can diffusely illuminate normal mapped world geometry with respect to an arbitrarily complex lighting environment in a few pixel shader instructions. Additionally, these same normal maps are used in other routines in our pixel shaders, such as those which compute Fresnel terms or access cubic environment maps for specular illumination.

7.3.2 World Specular Lighting

In keeping with our general philosophy of favoring complex global illumination and image based lighting techniques over the accumulation of a few simple local lights, we have chosen to use cubic environment maps for specular world lighting. In order to keep the memory cost of storing a large number of cube maps at a reasonable level, our level designers manually place point entities in their maps which are used as sample points for specular lighting. High dynamic range cube maps are pre-rendered from these positions using the Source engine itself and are stored as part of the static level data. At run time, world surfaces which are assigned a material requiring a specular term use either the cube map from the nearest sample point or may have one manually assigned in order to address boundary conditions.

7.4 Model Lighting

In order to light our characters and other models that can animate dynamically within our game worlds, we have employed a system that combines spatially varying *directional irradiance samples* from our radiosity solution with local light sources and environment mapping. On graphics hardware that supports 2.0 pixel shaders and lower, we compute diffuse illumination from up to two local light sources, either per pixel or per vertex. On newer graphics chips with longer pixel shaders, we can afford to include specular contributions with Fresnel terms as well as more custom effects.

7.4.1 Ambient Cube

At Valve, we heavily favor the look of global illumination in order to ground characters in our game worlds and make them seem to be truly present in their environment. The most significant component of *Half-Life 2* character and model lighting is the inclusion of a directional ambient term from *directional irradiance samples* precomputed throughout our environments and stored in an irradiance volume as in [Greger98]. While most games today still use a few local lights and a gray ambient term, there have been a few examples of other games including a more sophisticated ambient term. For example, characters in many light mapped games such as *Quake III* include an ambient term looked up from the light map of the surface the character is standing on [Hook99]. Most similarly to the Source engine, the game *Max Payne 2* stored linear 4-term spherical harmonic coefficients, which effectively reduce to a pair of directional lights, in an irradiance volume throughout their game worlds for use in their character lighting model [Lehtinen06].

Due to the particular representation of our irradiance samples (essentially low resolution cubic *irradiance environment maps* [Ramamoorthi01]) and due to the fact that they are computed so that they represent only indirect or bounced light, we call the samples *Ambient Cubes*. The evaluation of the ambient cube lighting term, which can be performed either per pixel or per vertex, is a simple weighted blending of six colors as a function of a world space normal direction as illustrated in the following HLSL routine.

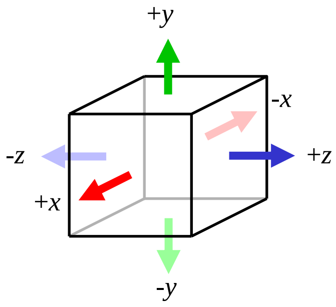
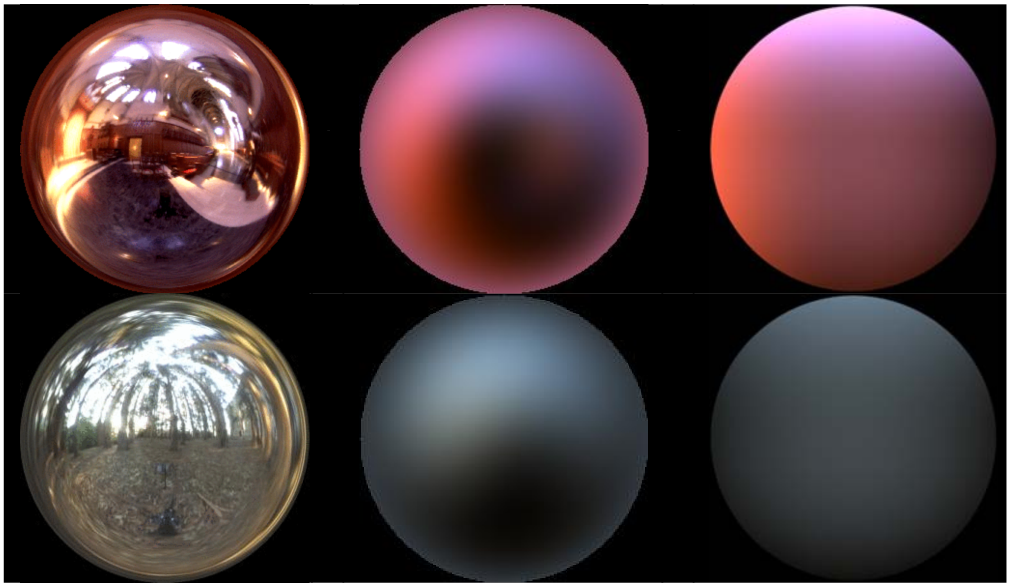


Figure 4*. Ambient cube basis*

```
float3 AmbientLight( const float3 worldNormal )
{
      float3 nSquared = worldNormal * worldNormal;
      int3 isNegative = ( worldNormal < 0.0 );
      float3 linearColor;
      linearColor = nSquared.x * cAmbientCube[isNegative.x]   +
                    nSquared.y * cAmbientCube[isNegative.y+2] +
                    nSquared.z * cAmbientCube[isNegative.z+4];
      return linearColor;
}
```

Listing 1*. Ambient Cube Routine (6 vertex shader instructions or 11 pixel shader instructions)*

The ambient cube is intuitive and easy to evaluate in either a pixel or vertex shader, since the six required colors can be loaded into shader constants. As mentioned above, we can afford to evaluate only two local diffuse lights on our target graphics hardware. In addition to this, any significant lights beyond the most important two are assumed to be non-local and are combined with the ambient cube on the fly for a given model. These six RGB colors are a more concise choice of basis than the first two orders of spherical harmonics (nine RGB colors) and this technique was developed in parallel with much of the recent literature on spherical harmonics such as [Ramamoorthi01]. As shown in Figure 5 below, we could potentially improve the fidelity of our indirect lighting term by replacing this term with spherical harmonics. With the increased size of the ps_3_0 constant register file (up to 224 from 32 on ps_2_*x* models) this may be a good opportunity for improvement in our ambient term going forward.



Original Light Probe

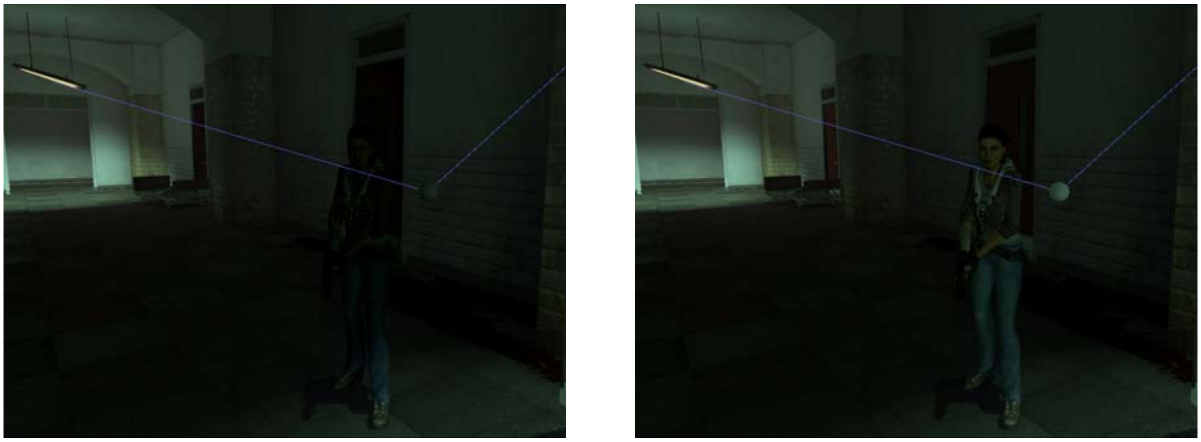
Irradiance environment maps from [Ramamoorthi01]

Irradiance environment maps represented using Valve ambient cube

Figure 5. *Comparison of spherical harmonics with Valve Ambient Cube*

7.4.2 Irradiance Volume

The *directional irradiance samples* or ambient cube sample points are stored in a 3D spatial data structure which spans our game environments. The samples are stored non-hierarchically at a fairly coarse (4’ x 4’ x 8’) granularity since indirect illumination varies slowly through space and we only need a fast lookup, not extreme accuracy. When drawing any model that moves in our environment, such as the character Alyx in the figure below, the nearest sample point to the model’s centroid is selected from the irradiance volume without any spatial filtering. Of course, this can result in hard transitions in the directional ambient component of illumination as the character or object moves throughout the environment, but we choose to time-average the ambient and local lighting information to limit the effect of lighting discontinuities, lessening the visual popping in practice. On important hero characters such as Alyx, we also heuristically boost the contribution of the directional ambient term as a function of the direct lights so that we don’t end up with the direct illumination swamping the ambient and causing a harsh appearance.



Without boosted ambient term

With boosted ambient term

***Figure 6**. Alyx illuminated with two distant direct lights and directional ambient term*

7.4.3 Local lighting

As mentioned above, the ambient cube data precomputed throughout our game worlds incorporates only bounced or indirect lighting. For direct lighting from local lights, we add in modified per-pixel Lambertian and Phong terms. On ps_2_0 video cards (20% of our users) we compute diffuse illumination for up to two local lights. Any nearby local lights beyond the two most significant lights are added into the ambient cube on the fly. Starting with *Half-Life 2: Episode 1*, on video cards which support ps_2_b or ps_3_0 shader models (40% of our users), the Source engine also computes Phong terms for the two most significant local lights.

7.4.4 Half Lambert Diffuse

In order to soften the diffuse contribution from local lights, we scale the dot product from the Lambertian model by ½, add ½ and square the result so that this dot product, the red cosine lobe in Figure 7 at right, which normally lies in the range of -1 to +1, is instead in the range of 0 to 1 and has a pleasing falloff (the blue curve in Figure 7). This technique, which we refer to as *Half Lambert* prevents the back side of an object with respect to a given light source from losing its shape and looking too flat. This is a completely non-physical technique but the perceptual benefit is enormous given the trivial implementation cost. In the original game *Half-Life*, where this diffuse Half Lambert term was the only term in most of the model lighting, it kept the characters in particular from being lit exclusively by a constant ambient term on the back side with respect to a given light source.

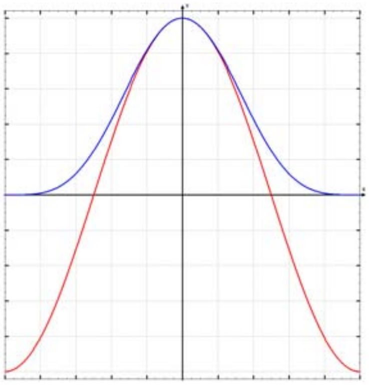
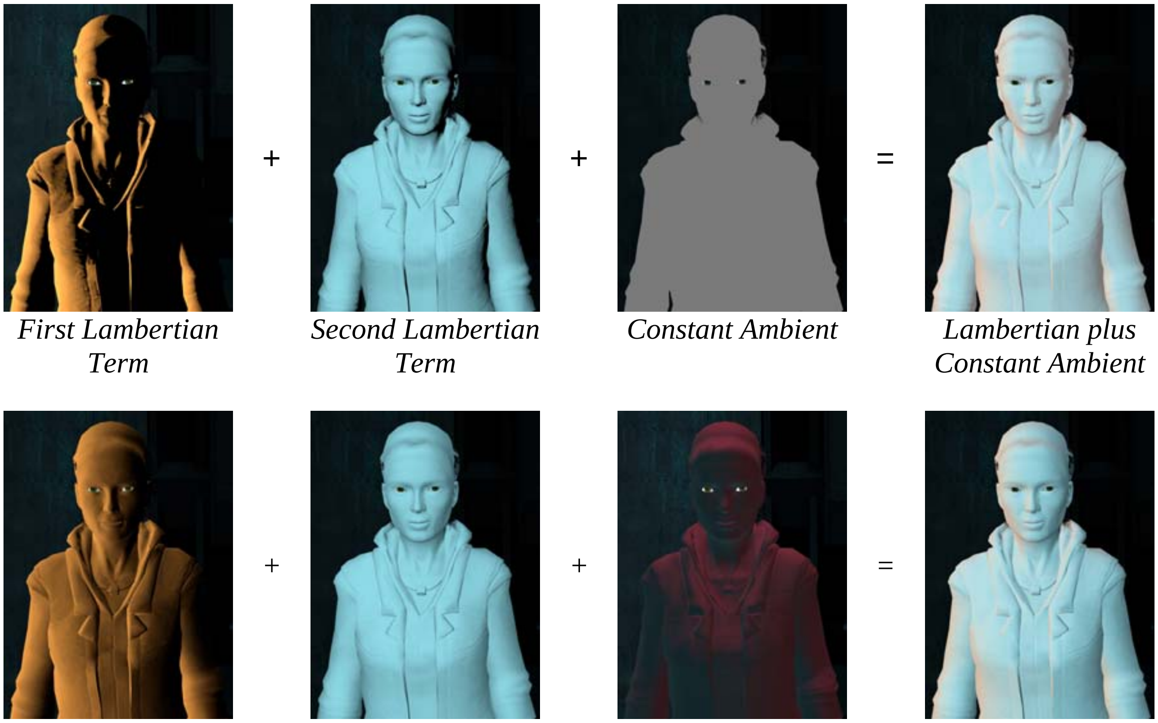


***Figure 7**. Cosine lobe (red) Half-Lambert function (blue)*



First Lambertian Term

Second Lambertian Term

Constant Ambient

Lambertian plus Constant Ambient

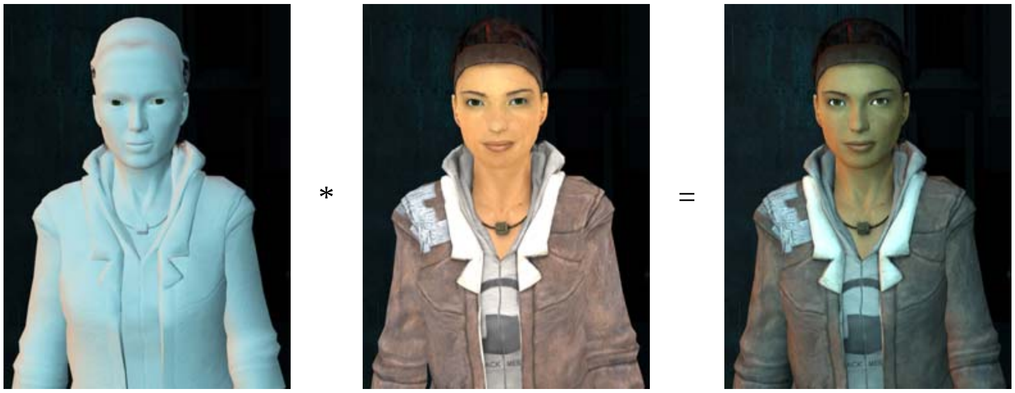
First Half Lambert Term

Second Half Lambert Term

Ambient Cube

Half Lambert and Ambient Cube

***Figure 8.** Comparison of traditional Lambertian and constant ambient terms with Half Lambert and Ambient Cube terms for diffuse lighting*

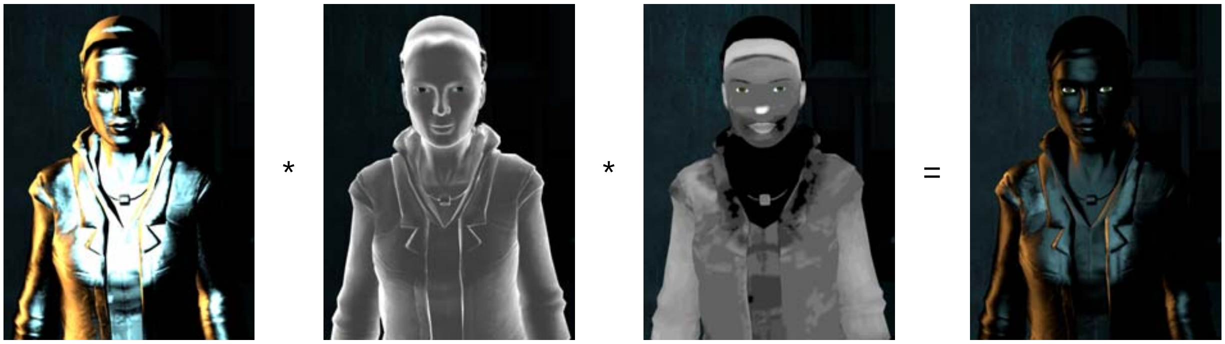


Diffuse Lighting from Figure 8 Above *Albedo* *Diffuse Lighting*

Figure 9. *Diffuse character lit by two Half-Lambert terms and ambient cube*

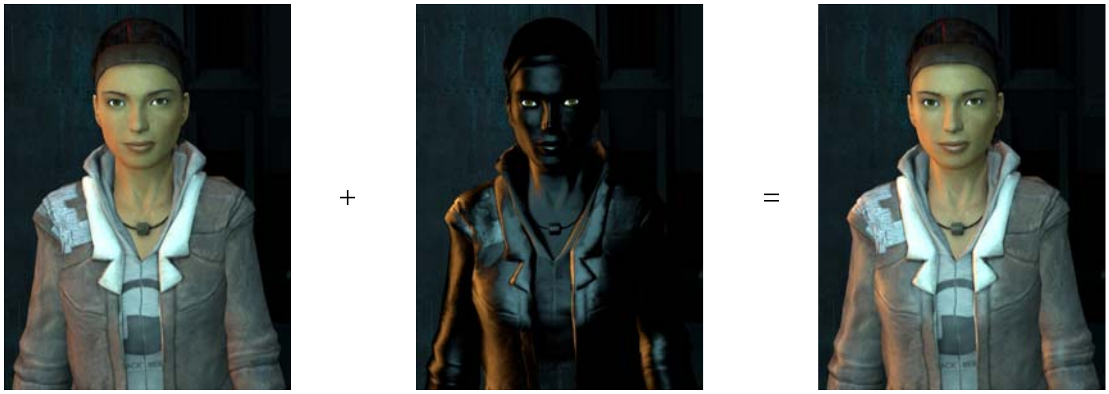
7.4.5 Local Specular

On pixel shader models ps_2_b and ps_3_0, our shading algorithms are no longer limited by shader length, so an easy way to increase visual quality is the inclusion of specular terms where appropriate. Starting with *Half-Life 2: Episode 1*, we have given our artists the ability to include Phong terms which can be driven by specular mask and exponent textures. As you can see in the images in Figure 11, this term integrates naturally with our existing lighting and provides greater definition, particularly to our heroine, Alyx Vance. In order to keep the specular terms from causing objects or characters from looking too much like plastic, we typically keep the specular masks (Figure 10c) fairly dim and also ensure that the specular exponents are low in order to get broad highlights. We also include a Fresnel term (Figure 10b) in our local specular computations in order to heavily favor rim lighting cases as opposed to specular highlights from lights aligned with the view direction. Under very bright lights, the specular rim highlights are often bright enough to bloom out in a dramatic fashion as a result of the HDR post-processing techniques which we will discuss in the following section.



(a) Two Phong Lights *(b) Fresnel Term* *(c) Specular Mask* *(d) Final Specular*

Figure 10*. Specular lighting in* Half-Life 2: Episode 1



(a) Diffuse from Figure 9 *(b) Specular from Figure 10* *(c) Final Lighting*

Figure 11. *Specular lighting term added for final character lighting equation. In* Half-Life 2: Episode 1, *60% of our current users will see the image in (a) while the 40% with hardware supporting ps_2_b and higher shaders will see the additional specular lighting in (c).*

7.5 High Dynamic Range Rendering

After shipping Half-Life 2, we implemented High Dynamic Range (HDR) rendering in the Source engine using a novel method which runs on graphics cards which support 2.0 pixel shaders and 16-bit per channel source textures [Green06]. This means that a whopping 60% of our users get to experience our games in HDR today. This technology was used to create the *Lost Coast* demo and has been used in all subsequent Valve games including *Day of Defeat: Source* and the recent episodic release *Half-Life 2: Episode 1*.

Many different methods for performing HDR rendering on modern GPUs were implemented and evaluated before finding the set of tradeoffs most appropriate for rendering our HDR content on current GPUs. Our implementation makes use of primarily 8-bit per channel textures with 16-bit per channel (either floating point or fixed point depending upon hardware support) light maps and cubic environment maps plus a few 16-bit per channel hand-authored textures for the sky, clouds, and sun.

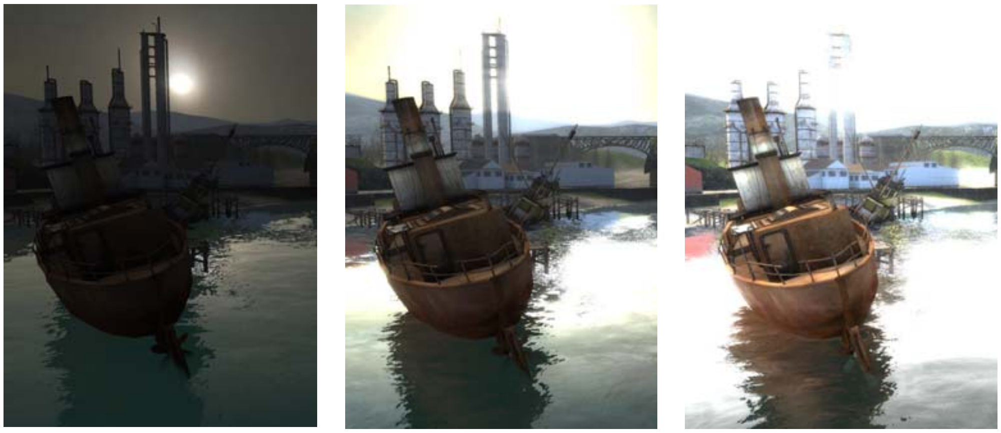


***Figure 10**. Scene from Lost Coast at Varying Exposure Levels*

Since all of the shaders in the Source engine's DirectX9 code path are single pass, it is possible to perform HDR lighting and tone mapping operations during normal rendering. This required the addition of a tone mapping subroutine to all pixel shaders but enabled us to render to ordinary 8-bit per channel render targets. By using 8-bit per channel render targets instead of floating point buffers, there is very little performance difference with the low dynamic range path, memory usage is the same and, most importantly, multisample anti-aliasing is supported. This also allows us to avoid an extra copy and conversion of the rendering output to the final display format. When intermediate renderings are used as the inputs for subsequent graphics operations, the dynamic range of these intermediate results is carefully managed so as to still provide high quality outputs from 8-bit data. For instance, renderings done for dynamic reflection in the water have their exposure adjusted by the water's reflectivity coefficient so that proper highlights from the sun are still produced.

When actual full-range HDR rendering outputs are needed (such as when rendering the cube maps used for reflection using the engine), the engine is used to produce a series of renderings at varying exposure levels, which are subsequently combined to produce one floating point HDR output image.

Auto exposure calculations needed for tone mapping are performed in a unique fashion in the Source engine. A pixel shader is applied to the output image to determine which output pixels fall within a specified luminance range and writes a flag into the stencil buffer indicating whether each pixel passed the test. An asynchronous occlusion query is then used to count how many pixels were within the specified range. When the occlusion query's results become available, they are used to update a running luminance histogram. One luminance range test is done per frame, amortizing the cost of generating the histogram over multiple frames. Time averaging is done to smooth out the results of this incremental histogram. This technique has a number of advantages over other methods, including the fact that it computes a full histogram instead of just an average luminance value, and that this histogram is available to the CPU without stalling the graphics pipeline or performing texture reads from GPU memory. In addition to the running histogram, other auto exposure and bloom filter parameters are controllable by level designers and can be authored to vary spatially with our game worlds or in response to game events. In addition to adjusting our tone mapping operations as a function of the running luminance of our scenes, we perform an image-space bloom in order to further stylize our scenes and increase the perception of brightness for really bright direct or reflected light. In the offline world, such bloom filtering is typically performed in HDR space prior to tone mapping [Debevec00] but we find that tone mapping before blooming is not an objectionable property of our approach, particularly considering the large number of users who get to experience HDR in our games due to this tradeoff.

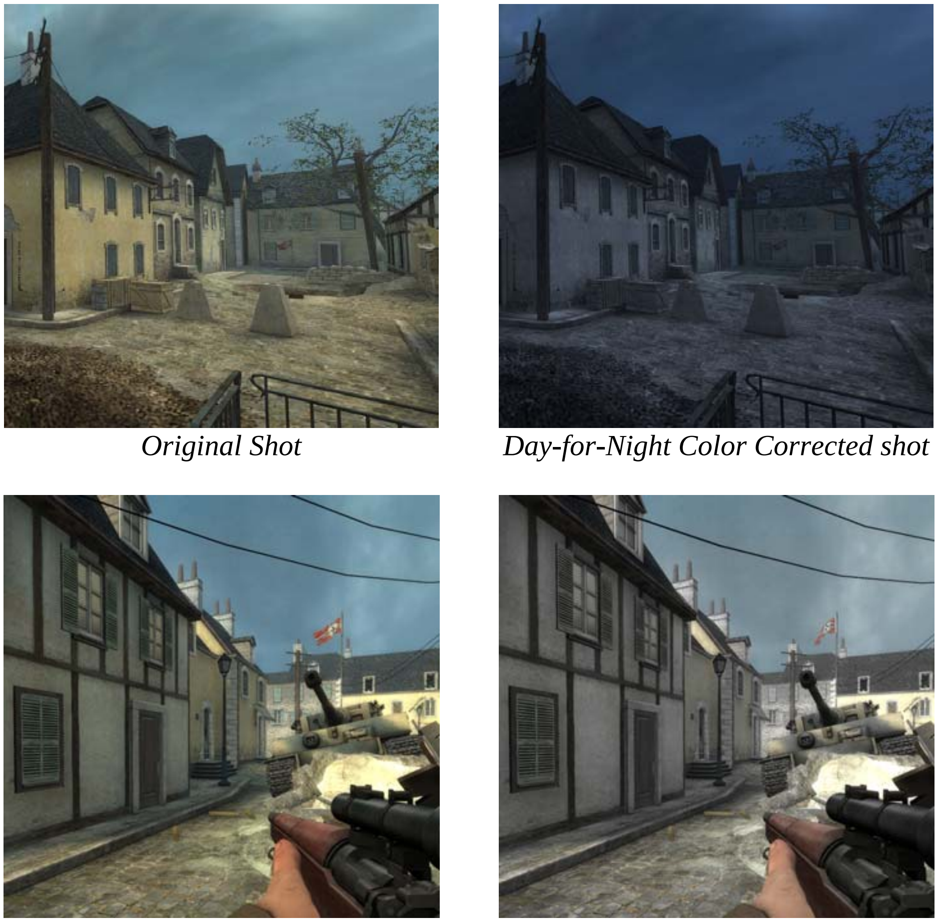
For those interested in implementation details, the source code for the auto exposure calculation is provided in the Source SDK.

7.6 Color Correction

Another important rendering feature, which we first shipped in the game *Day of Defeat: Source*, is real-time color correction. Human emotional response to color has been exploited in visual art for millennia. In the film world, the process of *color correction* or *color grading* has long been used to intentionally emphasize specific emotions. As it turns out, this technique, which is fundamentally a process of remapping colors in an image, can be very efficiently implemented on modern graphics hardware as a texture-based lookup operation. That is, given an input image, we can take the *r*, *g*, *b* color at each pixel and re-map it to some other color using an arbitrary function expressed as a volume texture. During this process, each pixel is processed independently, with no knowledge of neighboring pixels. Of course, image processing operations such as blurring or sharpening, which *do* employ knowledge of neighboring pixels, can be applied before or after color correction very naturally.

As mentioned in the preceding section on HDR rendering, we are already performing some image processing operations on our rendered scene. Including a color correction step fits very easily into the existing post-processing framework and decouples color correction from the rest of the art pipeline. We have found color correction to be useful for a variety of purposes such as stylization or to enhance a particular emotion in response to gameplay. For example, after a player is killed in our World War II themed game *Day of Defeat: Source*, the player can essentially look over the shoulder of any of his living team-mates for a short period before he is allowed to re-enter play. During this time, we use color correction and a simple overlaid film grain layer to give the rendering the desaturated look of WWII era archival footage.

Of course, color correction in real-time games can go beyond what is possible in film because it is possible to use game state such as player health, proximity to a goal or any other factor as an input to the color correction process. In general, the dynamic nature of games is both a curse and a blessing, as it is more difficult to tweak the look of specific scenes but there is far more potential to exploit games' dynamic nature due to the strong feedback loop with the player. We think of color correction as one of many sideband channels that can be used to communicate information or emotion to a player and we believe we are only just beginning to tap into the potential in this area.



Original Shot *Day-for-Night Color Corrected shot*

Original Shot *Desaturated with Color Correction*

***Figure 11**. Real-Time Color Correction in the Source Engine*

Beyond its use as an emotional indicator, color correction is a powerful tool for art direction. The amount of content required to produce a compelling game environment is rising rapidly with GPU capabilities and customer expectations. At the same time, game development team sizes are growing, making it impractical to make sweeping changes to source artwork during the later stages of production. With the use of color correction, an Art Director is able to apply his vision to the artwork late in the process, rather than filtering ideas through the entire content team. This is not only an enormous effort-multiplier, but it is invariably true that a game development team knows more about the intended player emotional response late in the process. Having the power to make art changes as late as possible is very valuable. If an art team authors to a fairly neutral standard, an art director is able to apply game-specific, level-specific or even dynamic event-specific looks using color correction. Another less obvious benefit which is particularly important to a company like Valve which licenses it engine and actively fosters a thriving *mod* community is the power that color correction provides for mod authors or Source engine licensees to inexpensively distinguish the look of their titles from other Source engine games.

For our games, a level designer or art director authors color correction effects directly within the Source engine. A given color correction function can be built up from a series of familiar layered operators such as curves, levels and color balance or can be imported from other popular software such as Photoshop, After Effects, Shake etc. Once the author is satisfied with their color correction operator, the effect can be baked into a volume texture that describes the color correction operation and is saved to disk. This color correction operator can then be attached to a node within our game world using our level design tool, *Hammer*. It is possible to attach a color correction function to a variety of nodes such as trigger nodes or distance based nodes, where the effect of the color correction operator decreases with distance from a given point. Essentially any game logic can drive color correction and several color correction operators can be active at once, with smooth blending between them.

7.7 Acknowledgements

Many thanks to James Grieve, Brian Jacobson, Ken Birdwell, Bay Raitt and the rest of the folks at Valve who contributed to this material and the engineering behind it. Thanks also to Paul Debevec for the light probes used in Figure 5.

7.8 Bibliography

Debevec, P. 2000. Personal Communication

Diefenbach, P. J. 1997. Multi-pass Pipeline Rendering: Realism For Dynamic Environments. Proceedings, 1997 Symposium on Interactive 3D Graphics.

Franke, S. 2006. Personal Communication

Green, C., and McTaggart, G. 2006. High Performance HDR Rendering on DX9-Class Hardware. Poster presented at the *ACM Symposium on Interactive 3D Graphics and Games*.

Greger, G., Shirley, P., Hubbard, P. M, and Greenberg, D. P. 1998. The Irradiance Volume. IEEE Computer Graphics & Applications, 18(2):32-43,

Hook, B. 1999. The Quake 3 Arena Rendering Architecture. Game Developer's Conference

Lehtinen, J. 2006. Personal Communication

McTaggart, G. 2004. Half-Life 2 Shading, GDC Direct3D Tutorial

Ramamoorthi, R. and Hanrahan, P. 2001. An Efficient Representation for Irradiance Environment Maps. SIGGRAPH 2001, pp. 497-500.

Chapter 8

Ambient Aperture Lighting

Chris Oat[12] and Pedro Sander[13]
ATI Research

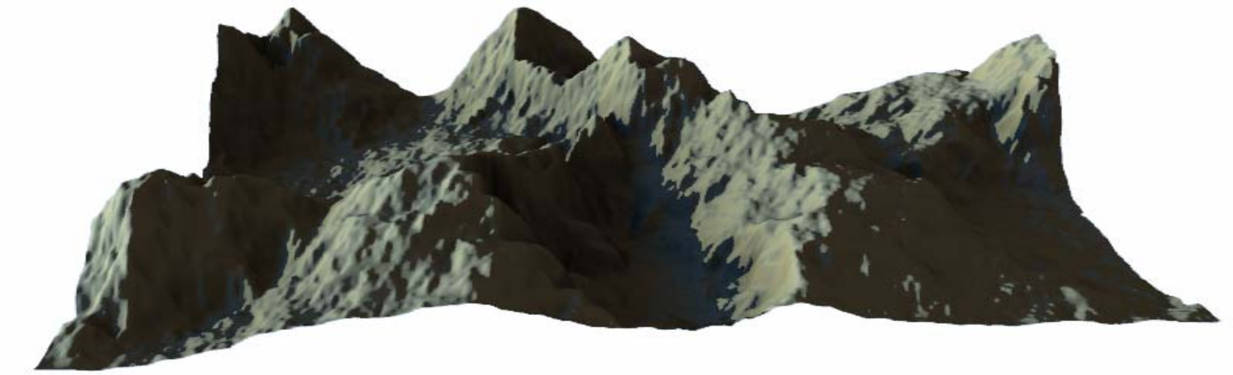


Figure 1: *A terrain is rendered in real-time and shaded using the Ambient Aperture Lighting technique described in this chapter.*

8.1 Abstract

A new real-time shading model is presented that uses spherical cap intersections to approximate a surface's incident lighting from dynamic area light sources. This method uses precomputed visibility information for static meshes to compute illumination, with approximate shadows, from dynamic area light sources at run-time. Because this technique relies on precomputed visibility data, the mesh is assumed to be static at render-time (i.e. it is assumed that the precomputed visibility data remains valid at run-time). The ambient aperture shading model was developed with real-time terrain rendering in mind (see Figure 1 for an example) but it may be used for other applications where fast, approximate lighting from dynamic area light sources is desired.

[12] coat@ati.com
[13] psander@ati.com

8.2 Introduction

Shadow mapping large terrains is frequently impractical for real-time applications because it requires transforming and rasterizing the terrain data multiple times per frame. Additionally, shadow maps are ill suited for terrain rendering as they don't allow for area light sources such as the sun and the sky. Finally, shadow maps exhibit aliasing artifacts particularly when used with large scenes such as terrains. Horizon mapping techniques for rendering self-shadowing terrains in real-time only allow point or directional light sources [Max88]. Precomputed Radiance Transfer allows for area light sources but assumes low-frequency lighting environments [Sloan02].

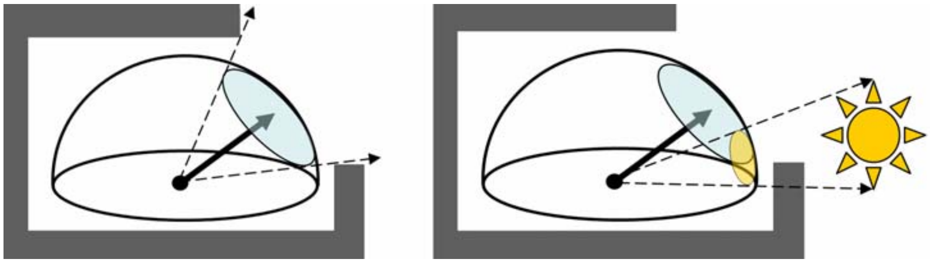


Figure 2: *[Left] A spherical cap is used to approximate a contiguous region of visibility (an aperture) on a point's upper hemisphere. [Right] The visible region acts as an aperture which restricts light such that it only reaches the point from visible (unoccluded) directions.*

Our approach allows for object self-shadowing from dynamic area light sources with the caveat that the object is not deformable (only rigid transformations are supported). Our technique stores a small amount of precomputed visibility data for each point on the mesh. The visibility data approximates contiguous regions of visibility over a point's upper hemisphere using a spherical cap (see Figure 2). This spherical cap acts as a circular aperture for incident lighting. The aperture is oriented on the hemisphere in the direction of average visibility and prevents light originating from occluded directions from reaching the point being rendered. At render time, area light sources are projected onto the point's upper hemisphere and are also approximated as spherical caps. Direct lighting from the area light source is computed by determining the area of intersection between the visible aperture's spherical cap and the projected area light's spherical cap. The overall diffuse response at the point is computed by combining the area of intersection with a diffuse falloff term described later.

The algorithm comprises of two stages. In a preprocessing stage, a contiguous circular region of visibility is approximated for each point on the surface of the model (Section 8.3). The data can be computed per texel and stored in a texture map, or per vertex and incorporated into the model's vertex buffer. Then, at rendering time, a pixel shader computes the amount of light that enters this region of visibility, and uses this result to shade the model (Section 8.4).

8.3 Preprocessing

Given a point – represented by a vertex on a mesh or by a texel in texture space – we wish to find its approximate visible region. The visible area at a point **x** is found by integrating a visibility function over the hemisphere. The visibility function evaluates to 1 for rays that don't intersect the scene and 0 otherwise. The percentage of visibility is then scaled by the area of a unit hemisphere and results in the solid angle of the visible region:

$$VisibleArea(x) = 2\pi \int_{\Omega} V(x,\omega)\,d\omega$$

This visible area is used as our aperture area. We do not store the area directly but instead store the arc length of a spherical cap of equivalent area (since this is what we need at run-time):

$$SphericalCapRadius = a\cos\left(\frac{-area}{2\pi} + 1\right)$$

The spherical cap's radius is a single floating point number that must be stored with the mesh (either per-vertex or per-texel).

The visible region's orientation on the hemisphere is determined by finding the average direction for which the visibility function evaluates to 1 (a.k.a. bent normal):

$$VisibleDir(x) = \int_{\Omega} V(x,\omega)\,\omega\,d\omega$$

This gives us an average direction of visibility and serves as our aperture's orientation on the surrounding hemisphere. This direction is stored as three floating point values (a three component vector) either per-vertex or per-texel.

8.4 Rendering

First we describe our rendering algorithm which, for each pixel, seeks to compute the intersection between the precomputed circular visibility region and the circular light source (projected area light). Then, we describe an optimization which significantly reduces computation time while achieving nearly identical qualitative results.

8.4.1 Intersecting Spherical caps

At rendering time, the spherical area light sources are projected onto the hemisphere and the radius of the light's enclosing spherical cap is computed. The amount of light that reaches a point is determined by computing the area of intersection between the precomputed spherical cap representing the aperture of visibility and the spherical light source as shown in Figure 3.

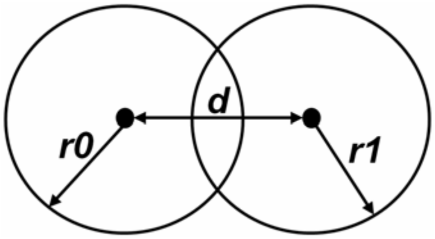


Figure 3: *The intersection area of two spherical caps is a function of the arc length of their radii (**r0**, **r1**) and arc length of the distance (**d**) between their centroids.*

The area of intersection for two spherical caps is computed as follows:

$$\begin{cases} 2\pi - 2\pi\cos(\min(r1, r0)) & \min(r1, r0) \le \max(r1, r0) - d \\ 0 & r0 + r1 \le d \\ \text{IntersectArea}(r0, r1, d) & \text{otherwise} \end{cases}$$

The first case, when ***min(r0,r1) <= max(r0,r1)-d*** occurs, the caps are fully intersected. This means that one cap is entirely inside of the other cap (or both caps are the same size and are perfectly aligned) and so we know the intersection area must be equal to the area of the smaller of the two caps. The next case occurs when the caps are far enough away from each other that we know that no intersection can possible occur (***r0+r1 <= d***), in this case the intersection area must be zero. Finally, if neither of the previous cases apply, there must be some partial intersection occurring so we must solve for the intersection area using our `IntersectArea` function which solves the following:

$$\begin{aligned} &2\cos(r1)\arccos\left(\frac{-\cos(r0)+\cos(d)\cos(r1)}{\sin(d)\sin(r1)}\right) \\ &-2\cos(r0)\arccos\left(\frac{\cos(r1)-\cos(d)\cos(r0)}{\sin(d)\sin(r0)}\right) \\ &-2\arccos\left(\frac{-\cos(d)+\cos(r0)\cos(r1)}{\sin(r0)\sin(r1)}\right) \\ &-2\pi\cos(r1) \end{aligned}$$

which is a simplified form of the result from [Tovchigrechko01]. Once the area of intersection is found, the net diffuse response is approximated by scaling the area of intersection by a Lambertian coefficient (cosine of the angle between the intersection centroid and the geometric surface normal, explained in section 8.4.3).

8.4.2 Optimization

The spherical cap intersection function (`IntersectArea`) may be too expensive to solve directly for many mid-range GPUs. Instead of solving this expensive function directly it may be desirable to use a less expensive approximation. Because the intersection function exhibits a smooth falloff with respect to increasing cap distance (see Figure 4) a `smoothstep` function is an appropriate approximation:

$$(2\pi - 2\pi\cos(\min(r1, r0)))\ \ \mathrm{smoothstep}\left(0,1,1-\frac{d-|r0-r1|}{r0+r1-|r0-r1|}\right)$$

The `smoothstep` function here returns 1.0 when a full intersection occurs and 0.0 when no intersection occurs. The result of the `smoothstep` is then scaled by the area of the smaller of the two spherical caps so that the end result is a smooth transition between full intersection and no intersection. Empirically, we did not notice any significant qualitative difference between using the `smoothstep` approximation and the actual intersection function. Figures 5 and 6 compare the exact intersection function and its approximation for both small and large circular light source radii. Note that the differences, which can only occur in the penumbra region, are imperceptible.

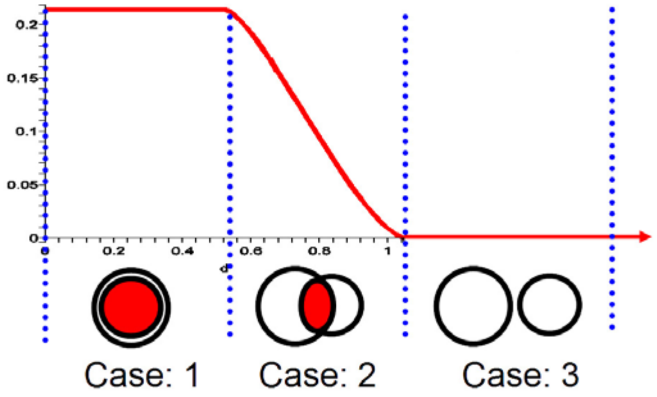
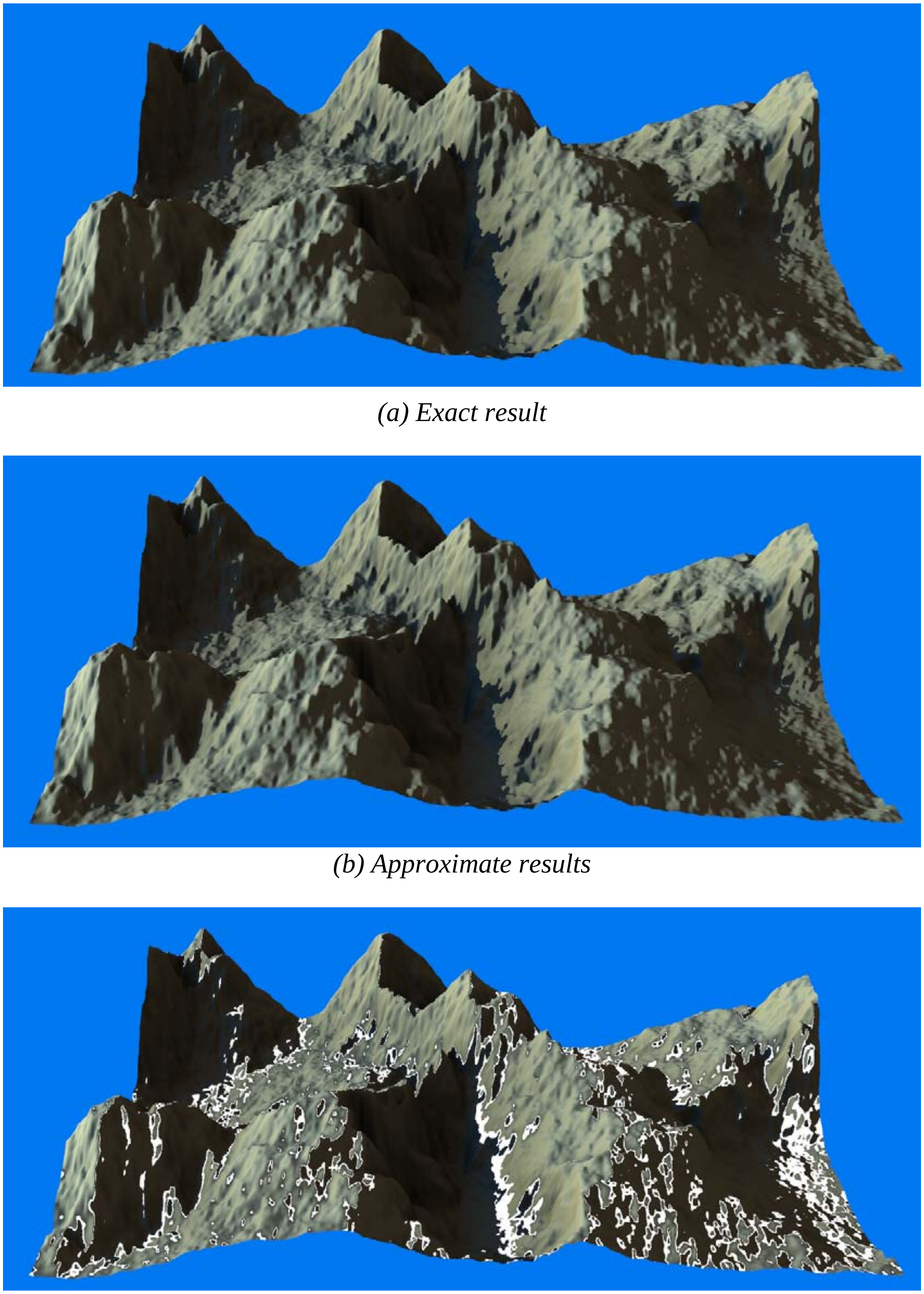


Figure 4: *Spherical cap intersection as a function of distance between spherical cap centers. Case 1: One cap entirely overlaps the over, full intersection occurs. Case 2: The full intersection function is evaluated to find the area of intersection between partially overlapping spherical caps. Case 3: The caps are too far apart for any intersection to occur, their intersection area is zero.*

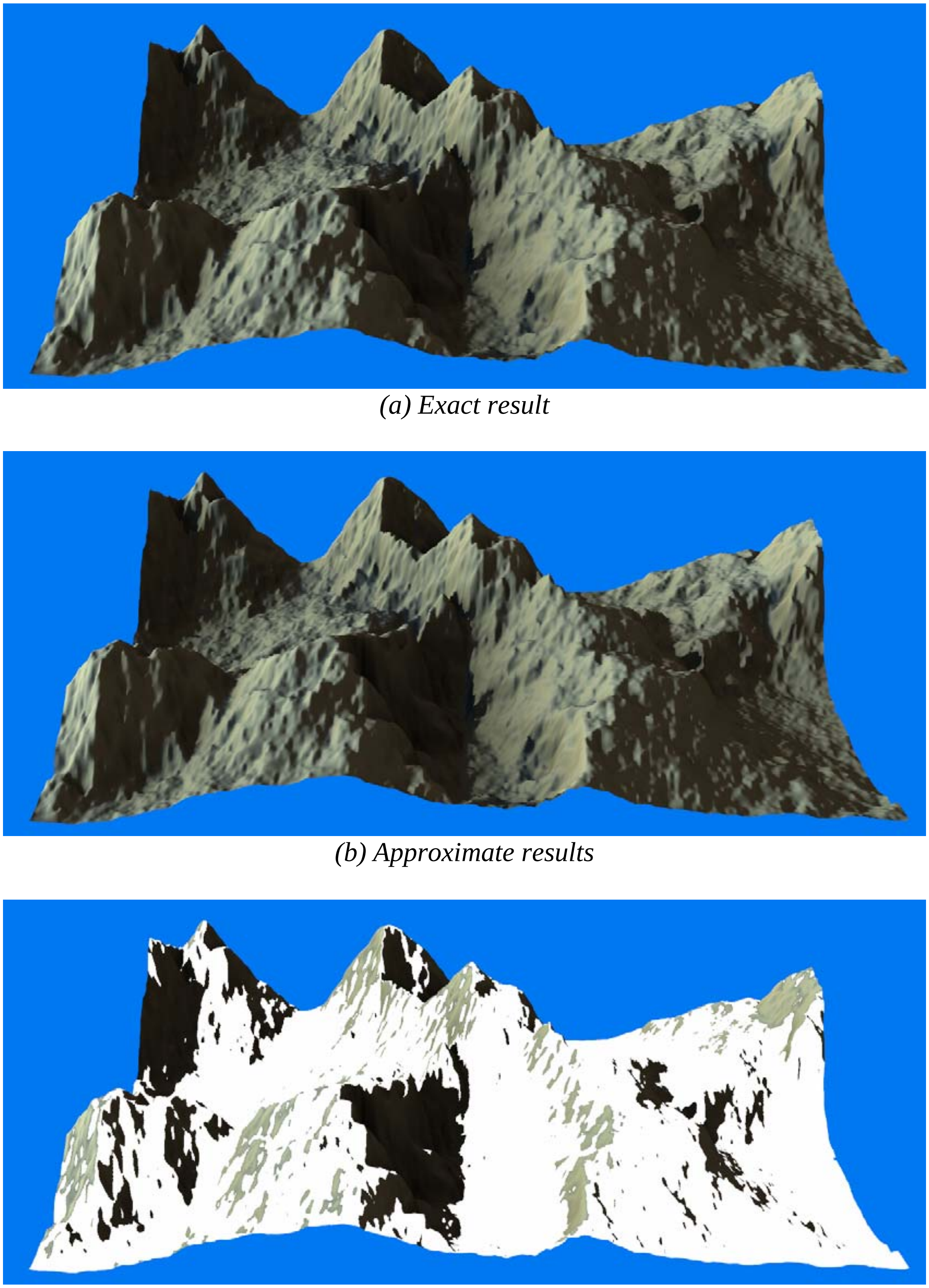


(a) Exact result

(b) Approximate results

(c) Penumbra region color-coded in white

Figure 5: *Rendering results using a light source with a small radius.*



(a) Exact result

(b) Approximate results

(c) Penumbra region color-coded in white

Figure 6: *Rendering results using a light source with a large radius.*

```
// Approximate the area of intersection of two spherical caps
// fRadius0 : First cap’s radius (arc length in radians)
// fRadius1 : Second caps’ radius (arc length in radians)
// fDist : Distance between caps (radians between centers of caps)
float SphericalCapIntersectionArea(float fRadius0, float fRadius1, float fDist)
{
   float fArea;

   if ( fDist  <= max(fRadius0, fRadius1) - min(fRadius0, fRadius1) )
   {
      // One cap in completely inside the other
      fArea = 6.283185308 - 6.283185308 * cos( min(fRadius0,fRadius1) );
   }
   else if ( fDist >= fRadius0 + fRadius1 )
   {
      // No intersection exists
      fArea = 0;
   }
   else
   {
      // Partial intersection exists, use smoothstep approximation
      float fDiff = abs(fRadius0 - fRadius1);
      fArea=smoothstep(0.0,
                       1.0,
                       1.0-saturate((fDist-fDiff)/(fRadius0+fRadius1-fDiff)));
      fArea *= 6.283185308 - 6.283185308 * cos( min(fRadius0,fRadius1) );
   }

   return fArea;
}
```

Listing 1: *An HSLS function that computes the approximate area of intersection of two spherical caps using our smoothstep approximation for partially overlapping spherical caps.*

8.4.3 Our Friend Lambert

Knowing the area of intersection between the aperture’s spherical cap and the light source’s spherical cap isn’t enough to compute lighting at the point, it really only tells us how much unoccluded light arrived at the point we’re rendering. A given area of intersection will result in more or less diffuse reflection depending on its orientation relative to the point’s surface normal. In other words, we must take Lambert’s Cosine Law into account. The correct way to handle this would be to convolve the intersection area with a cosine kernel, but this would require solving an integral and is too expensive (if we wanted to solve integrals, we wouldn’t be mucking about with spherical caps in the first place!). Instead, we account for the Lambertian falloff by first finding a vector from the point we’re shading to the center of the area of intersection (i.e. the intersection region’s centroid). A vector to the intersection region’s centroid is estimated by averaging the aperture’s orientation vector with the light’s vector. We then compute the dot product between this vector and the point’s geometric normal. This dot product is used to scale the incoming lighting and it results in a Lambertian falloff as the area of intersection approaches the horizon. For this approximation to be correct, we must assume that the area above the intersection region's centroid is about the same as the area below the intersection region's centroid (see figure 7).

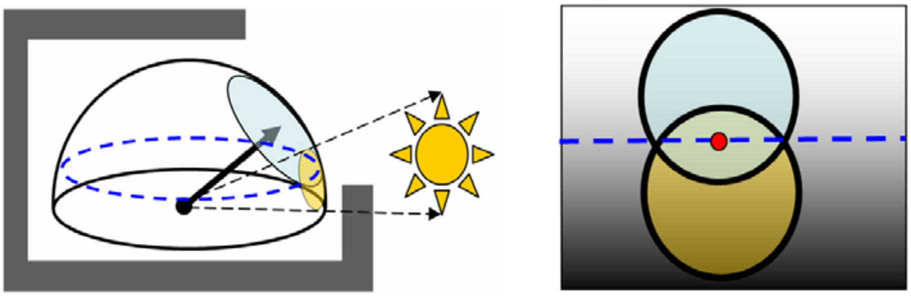


Figure 7: *[Left] The aperture's spherical cap is intersected with a light source's spherical cap. [Right] A rectangular plot of the two spherical caps along with the Lambertian falloff function (i.e. the cos(theta) term). Our approximation assumes that the intersection area above the intersection region's centroid (red) is the same as the intersection area below the region's centroid.*

This gives us a nice approximation of direct, diffuse lighting on our mesh but it doesn't handle indirect lighting at all and thus we require some additional terms to account for ambient lighting.

8.4.4 Ambient Light

What if light occupies only a portion of our aperture's area? Or none of the aperture's area? For outdoor lighting scenarios we should still get some indirect light scattering into our aperture and this needs to be accounted for in our shading model. The indirect sky light can be handled by filling the empty portion of our aperture with ambient light from the sky. Once the area of the light/aperture intersection is found, we can subtract this area from the aperture's total area to figure out how much of the aperture is unoccupied by the sun. This empty space can then be filled with indirect, ambient light. For outdoor rendering, we can use the average sky color for our ambient light. The average sky color could be found by sampling the lowest MIP level of a sky dome, for example, or it could just be derived based on time of day:

- Sky blue during the day
- Red or pink at sun set
- Black or deep blue at night

This works much nicer than the standard constant ambient term since it only applies to areas that aren't being lit directly and aren't totally occluded from the outside world. This method's advantage is that it doesn't destroy scene contrast by adding ambient light in areas that really should be dark because they're mostly occluded from the outside world.

8.5 Limitations

Ambient aperture lighting makes a lot of simplifying assumptions and approximations in order to reduce lighting computations when rendering with dynamic area light sources. There are certain cases where the assumptions break down and the shading model produces artifacts such as incorrect shadowing. One such assumption is that the visible region for any point on a mesh is both contiguous and circular. This is generally true for things like terrains where a point's visible region is more-or-less the horizon but it fails to handle the case of, for example, a sphere over a plane (see Figure 8).

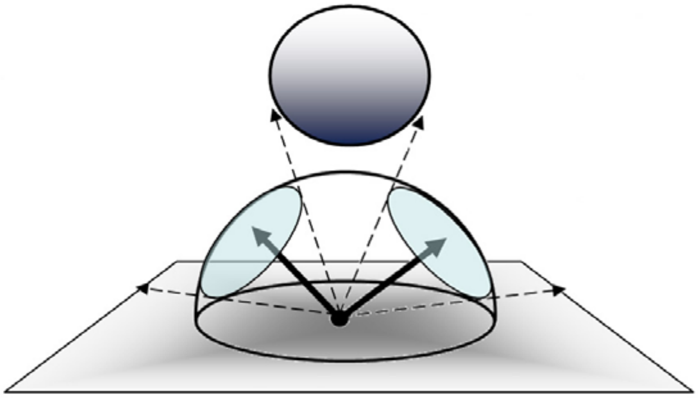


Figure 8: *Which way should the visibility aperture point? The visible region here is a band around the horizon which can't be closely approximated by a spherical cap.*

8.6 Conclusion

A real-time shading model that allows for dynamic area light sources was presented. This technique is well suited for outdoor environments lit by dynamic spherical area light sources (such as the sun) and is particularly useful for applications where fast terrain rendering is needed. Ambient aperture lighting makes several simplifying assumptions and mathematical approximations to reduce the computational complexity and storage costs associated with other real-time techniques that allow for dynamic area light sources.

8.7 Bibliography

MAX, N. L. *Horizon Mapping: Shadows for Bump-mapped Surfaces*. The Visual Computer 4, 2 (July 1988), 109-117.

SLOAN, P.-P., KAUTZ, J., SNYDER, J., *Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments*, SIGGRAPH 2002.

TOVCHIGRECHKO, A. AND VAKSER, I.A. 2001. *How common is the funnel-like energy landscape in protein-protein interactions?* Protein Sci. 10:1572-1583

Chapter 9

Fast Approximations for Global Illumination on Dynamic Scenes

Alex Evans[14]
Bluespoon

9.1 Abstract

An innovative lighting algorithm is presented that allows scenes to be displayed with approximate global illumination including ambient occlusion and sky-light effects at real-time rates. The method is scalable for high polygonal scenes and requires a small amount of pre-computation. The presented technique can be successfully applied to dynamic and animated sequences, and displays a striking aesthetic style by reducing traditional constraints of physical correctness and a standard lighting model.

9.2 Introduction

The introduction of pixel and fragment level programmable commodity GPU hardware in 2002 led to a significant shift in the way in which real-time graphics algorithms were researched and implemented. No longer was it a case of exploiting a few simple fixed function operations to achieve the desired rendered image – instead programmers could explore a wealth of algorithms, along with numerous variations and tweaks.

One specific class of algorithms of particular interest in the games industry, are those real-time algorithms which take as their target an art director's 'vision', rather than a particular subset of the physics of light in the real world.

These algorithms opened up the 'field of play' for graphic engine programmers, and greatly increased the opportunity for each engine to differentiate itself visually in a crowded and (in the case of games), fiercely competitive market. [Evans04]

Many problems in computer graphics are not yet practical to compute in real-time on commodity hardware, at least in the general case. Instead, techniques that can balance

[14] alex@bluespoon.com

the computational load between run-time and pre-processing costs allow different trade-offs that are applicable in different rendering systems (for example, pre-computed radiance transfer (PRT) techniques, which allow complex light transport to be computed in a static scene, then cached in a variety of ways which allow novel views and/or re-lighting of a scene to be rendered interactively).

In these course notes, we will describe one such approximation algorithm. There are many points at which decisions were made between a number of different, potentially useful algorithmic options. It is this process of iteration and choice, made possible by GPU programmability, more than the particular end result, that is the emphasis of the following discussion.

9.3 Algorithm outline

The goal of the algorithm is to allow real-time rendering of dynamic scenes from a movable viewpoint, lit under some aesthetically pleasing approximation to 'sky lighting'. An example of a sky-lit scene is shown in Figure 1. The key requirements set by our fictional art director are:

- Dark 'contact' shadows where objects come into contact with each other
- Darker regions inside the creases and valleys of complex organic shapes – often called 'ambient occlusion'.
- Ability to handle multiple objects each of which can move both rigidly and preferably, freely deform.

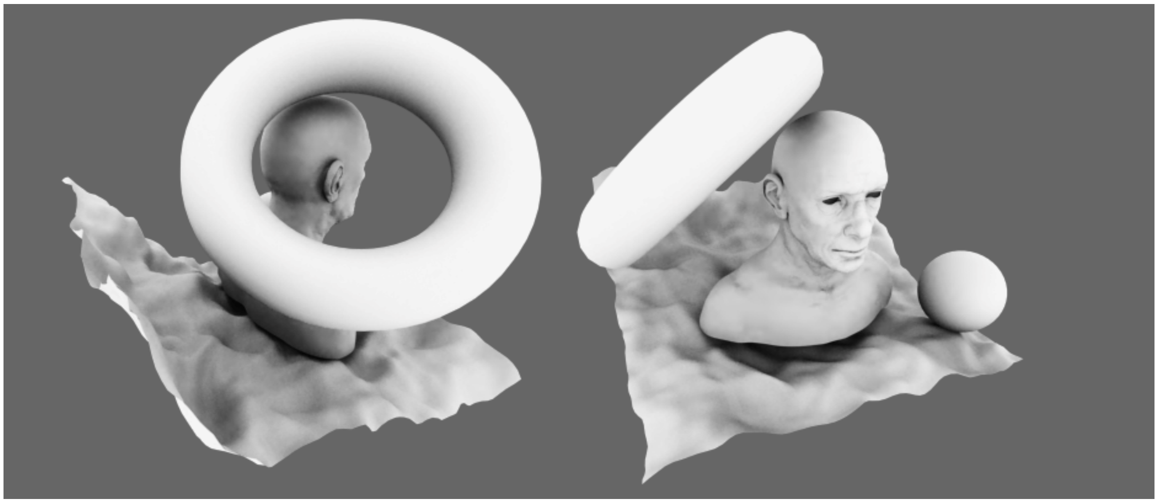


***Figure 1**. A detailed head model rendered using Autodesk's 3D Studio Max default renderer with a single skylight. (Model by Sebastian Schoellhammer, www.sebster.org)*

These kinds of effects are a key visual indicator in giving the impression of physical objects are in contact with each other, and have been widely investigated, dating back to the earliest days of graphics research. Path tracing [Kajiya86] gives a simple but very slow way of understanding and evaluating sky lighting: at each point P to be rendered, the fraction of the sky dome visible at that point is estimated by shooting a large number of rays towards the sky (considered to be an evenly light emitting hemisphere centered at the origin of the scene, of infinite radius). These rays bounce off any surfaces that they come across, until they are eventually considered to be absorbed or reach the sky dome itself. (Figure 2)

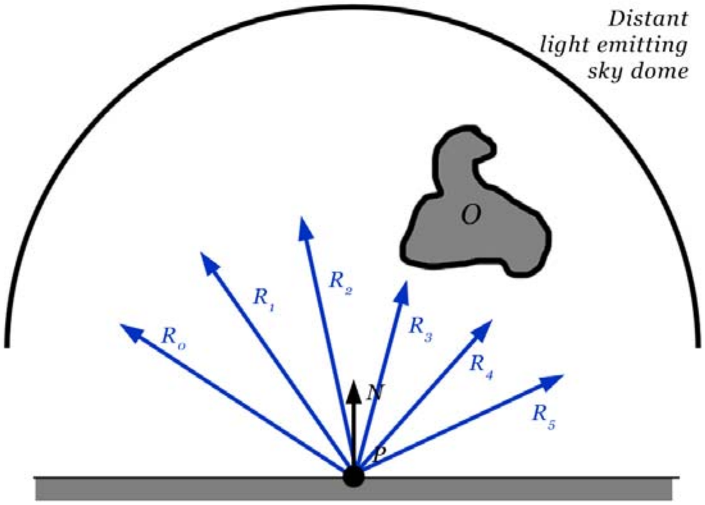


***Figure 2**. Path tracing a point P – by tracing rays from P towards a large number of points on the distant sky hemisphere, and measuring the fraction which make it to the sky, we can arrive at an estimate of the surface radiance of the point P as a result of the sky lighting. In the case illustrated, rays R_3 and R_4 are blocked by object O.*

By summing the results of a suitably large number of random rays, this Monte Carlo approach to solving the diffuse part of Kajiya's rendering equation eventually converges to an estimate of the surface radiance of the point P – and thus, when applied to all points visible to the eye, a nice-looking sky-lit scene. Ignoring ray bounces, this process amounts to computing an approximation to a simple integral giving the visibility of the sky in the hemisphere above P:

$$I_P = \int_{\Omega_P} V(\omega)B(v,\omega)d\omega$$

where I_P is the surface radiance of point P, Ω_P is the hemisphere above the surface at P, $B(v, \omega)$ is the BRDF of the surface being lit (including the diffuse cosine term, $max(N{\cdot}L,0)$), v is the direction towards the viewer from P, and $V(\omega)$ is a function defined such that $V(\omega)=1$ when rays in that direction reach the sky, and 0 when they do not.

Normally, V is the expensive term to compute, and thus the term that is normally chosen for various kinds of approximation by real-time rendering algorithms.

Many techniques exist which try to make the sky lighting problem more tractable by making various trade-offs. As such, real-time algorithms are often better characterized by the cases they can't handle than by the cases that they can. For example, many diffuse global illumination solutions rely on pre-computed representations of the geometry of the scene and the light flow around it. These include various forms of radiosity, pre-computed radiance transfer (PRT), irradiance caching, and table based approximations based on simplified occluders, such as spherical caps [Oat06], [Kontikanen05].

However, to illustrate the breadth of approach that programmable GPUs have given us, we are going to outline an unusual variant that has the following properties:

- Requires very little pre-computation and hence works well in dynamic scenes
- Has enormously large deviation from the accurate solution, but still looks aesthetically pleasing
- Allows limited bounce light effects without computational penalties

Note that we have to specify the following constraint for our algorithm: it will only support scenes that can be encompassed in a small volume. This is a direct result of the fact that the approach is based on the idea of signed distance fields (SDFs) which will be stored in a volumetric texture on the GPU. The above properties and constraints will provide us with a GPU-friendly data structure, whose computation is expensive but highly parallelizable, which encodes enough of the geometric layout of the scene that we can render approximate sky-lighting efficiently.

9.4 Signed Distance Fields

In a given scene consisting of solid bodies, a signed distance field is a simple scalar function S(P) defined at every point P in a (2D or 3D) space, such that

$S(P) = 0$ when it is on the surface of a body
$S(P) > 0$ when it is inside any body
$S(P) < 0$ when it is outside all bodies

Its magnitude is the minimum distance from that point to the surface of any body. A simple example of a 2D SDF is visualized in Figure 3.

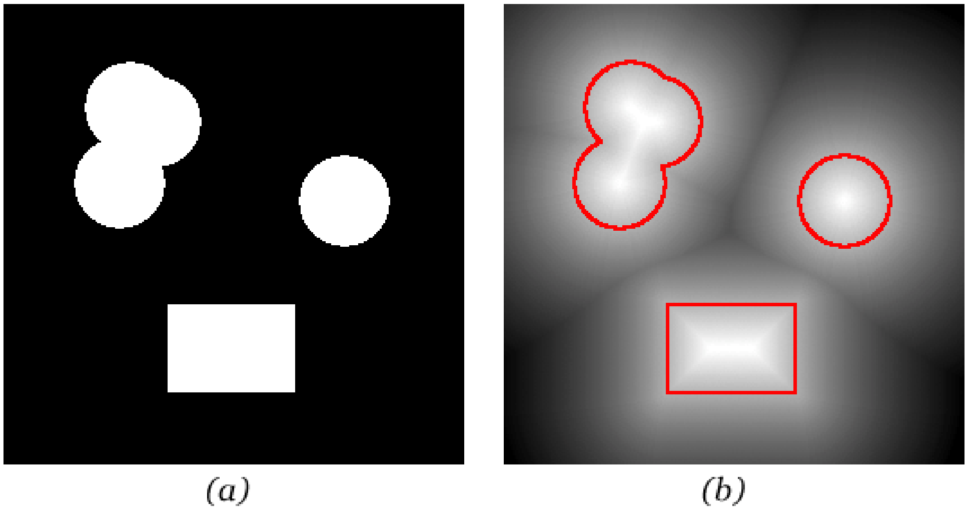


Figure 3. *a) shows a binary image representing a 2D scene, with points covered by an object marked white. b) Shows the SDF computed from (a) using 2D Euclidian distances, with -1 to 1 mapped to a grayscale gradient from black (-1) to white (1). The boundary of the objects (where SDF=0) is marked in red for clarity.*

Computing the SDF of an arbitrary polygonal 2D or 3D scene on a regularly sampled grid is a computationally expensive process, often approximated by repeated application of small (3x3x3) window-size non-linear filters to the entire volume [Danielson00, Grevera04]. We will deal with efficient generation of the SDF on the GPU later, but for the time being it's sufficient to give the motivation for computing it: in an approximate sense, it will serve as our means of rapidly computing the visibility of the sky from any point in the scene we wish to light.

Signed distance functions find application in several areas: they are an example of a 'potential function' which can rendered directly or indirectly using ray marching, level sets or marching cube techniques – for example to help accelerate the anti-aliased rendering of text and vector graphics [Pope01, Frisken02, Frisken06]. They are also useful in computer-vision in helping to find collisions, medial axes and in motion planning [Price05].

Signed distance fields can be computed analytically for simple shapes – and we shall be returning to this shortly – or tabulated in grids, octrees, or other spatial data structures. In this course we'll be using the simplest mapping to a GPU – namely an even sampling of the SDF over a volume, storing the values of the function in a volume texture. This kind of direct GPU representation is also used for a particular type of efficient GPU ray tracing called 'sphere tracing' – as used in the implementation of per-pixel displacement mapping in [Donnely05].

9.5 Curvature of a surface as an approximation to ambient occlusion

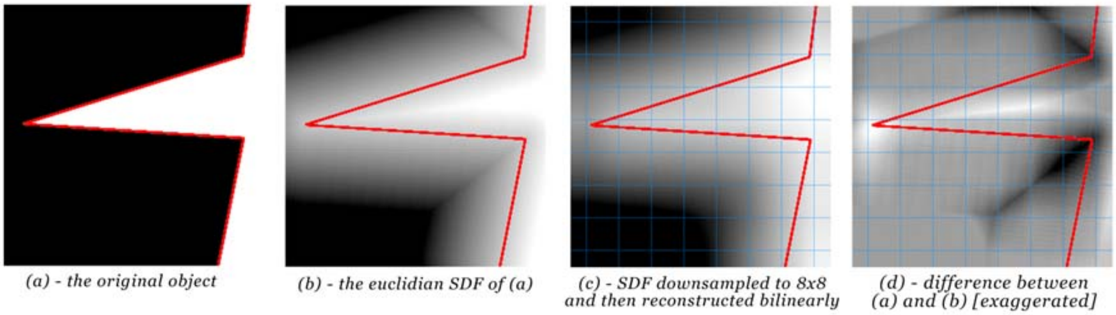

(a) - the original object
(b) - the euclidian SDF of (a)
(c) - SDF downsampled to 8x8 and then reconstructed bilinearly
(d) - difference between (a) and (b) [exaggerated]

***Figure 4**. (a) & (b) shows a part of a concave 2D object with its SDF. (c) shows the smoothing effect of sampling the SDF on a regular grid and then reconstructing it bilinearly. (d) Shows the difference between the smoothed and unsmoothed versions of the SDF. For points on the surface, the difference is related to the curvature of the surface – positive near concave corners (creases) such as the apex at the left, and negative near convex corners.*

The property we wish to make use of here is to do with using the SDF to find an approximation to the curvature of the surfaces of meshes in a scene [Masuda03]. In particular, we make the observation that in order to create the visual effect of darkening inside creases and crevices; we could darken the mesh wherever it has significant 'concavity' – inside wrinkles, behind ears, and so on – to simulate the effect of the sides of the concave region occluding the sky light. Figure 4 shows an example of how a regularly sampled SDF can help us – around a sharp concave feature, the SDF itself is also sharp and concave (Figure 4b). However, the errors introduced by the process of sampling the SDF on a regular, coarse grid and then reconstructing it using a simple trilinear filter have a smoothing effect (Figure 4c). A shader which sampled the SDF on the surface would expect the answer '0' if the SDF representation was error-free; however in the case of a blurred SDF, the result will be related to the curvature of the surface – slightly positive in concave regions and slightly negative in convex regions (Figure 4d).

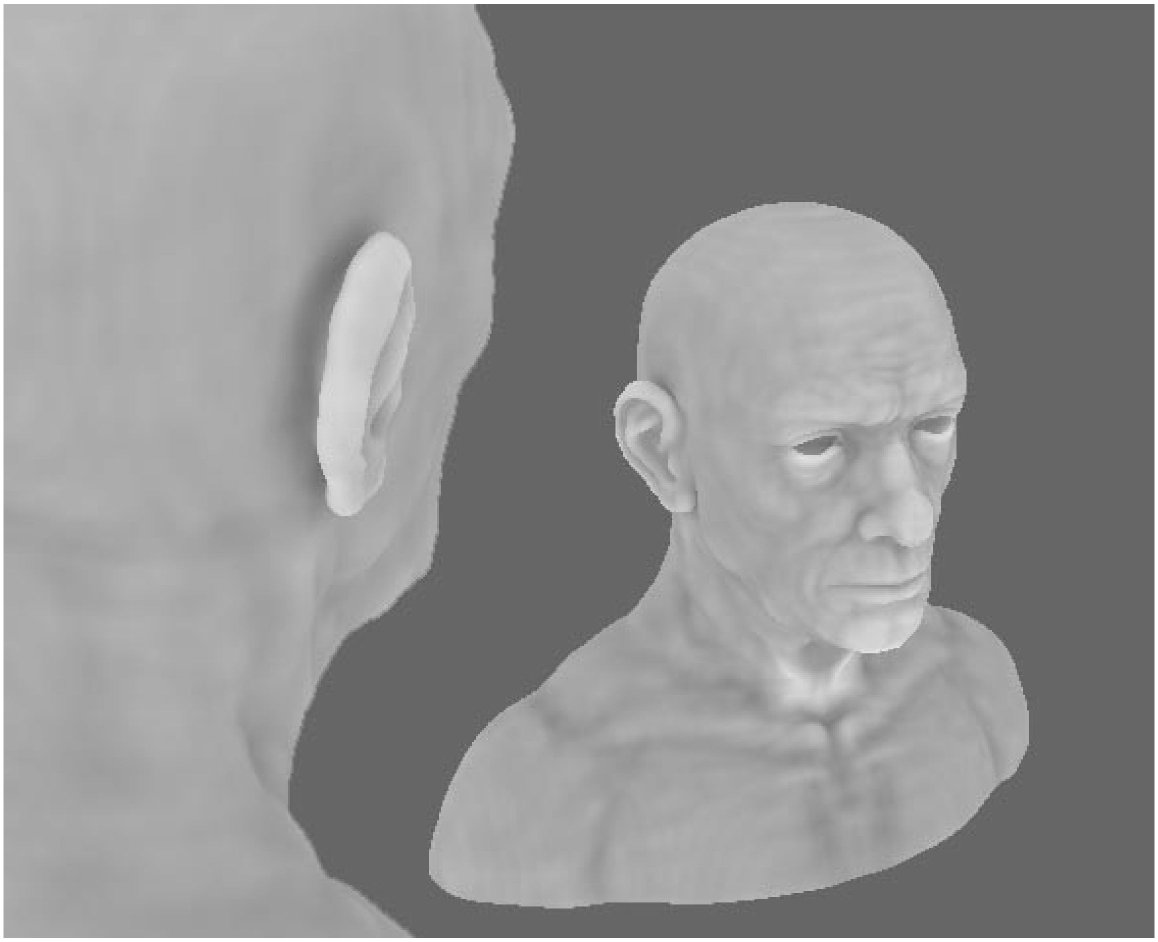


Figure 5. *The result of visualizing the error in the sampled SDF, stored as an 8 bit per pixel 64x64x64 volume texture. (Two views).*

Figure 5 shows the result of visualizing this effect directly. The SDF of the mesh was pre-computed on the CPU once and uploaded as a 64x64x64 volume texture to the GPU. The mesh was then rendered with a simple shader that sampled the volume at the point to be lit (on the surface of the mesh). The output colour was $0.5+exp(k * S)$ where k sets the overall contrast, and S is the value of the SDF texture sampled at the rendered point. While Figure 5 doesn't yet look much like Figure 1, it shows some promise and exhibits the desired 'creases are dark' ambient occlusion aesthetic look, for example, behind the ears.

Point sampling the SDF and then reconstructing it trilinearly introduces high frequency aliasing, visible in the output (and Figure 5) as slight banding across the mesh surface. These artifacts can be greatly reduced by pre-filtering the SDF by low-pass filtering it with (for example) a separable Gaussian low-pass filter. An additional advantage of this pre-filtering step is that the width of the Gaussian filter allows us to effectively choose the characteristic scale of the concavities that the algorithm highlights. We will use this feature in the next section.

It should be noted that the distance encoded in the signed distance function may be measured in a number of different ways – Euclidian, 'Manhattan', chess-square etc., and may be exact or approximate. For the particular application of estimating curvature of a mesh outlined in these course notes, it turns out that the particular distance metric chosen isn't particularly important. The only required property of the 'pseudo SDF' is that we can estimate curvature in the manner outlined in Figure 4. Any function which

- can be quickly computed
- is positive inside objects, and negative outside
- decreases monotonically as the sample point moves away from object surfaces

It turns out that even a heavily blurred version of the binary image (eg blurring Figure 3a directly) satisfies these conditions; however the underlying concept of an SDF provides us with an intuitive theoretical basis for experimenting with different variations. This is simply another example of the breadth of experimentation that art-driven programmability affords us – the 'correct' choice of distance metric or blur kernel can only be guided by the particular 'look' required.

9.6 Achieving the skylight look via SDFs

Only one more observation is required to achieve convincing sky lighting using our SDF representation of the scene. We wish to incorporate sky shadowing effects on any point P, from *all* objects in the scene, whatever their distance from the point P, not just the effect of nearby concavities. Objects at a greater distance should have a lesser, 'blurrier' effect than those near the point P. Since the focus here is on achieving a desired visual affect without worrying too much about the physicality of the situation, we look to our SDF to see if there is some way that it can help us express these broader scale interactions.

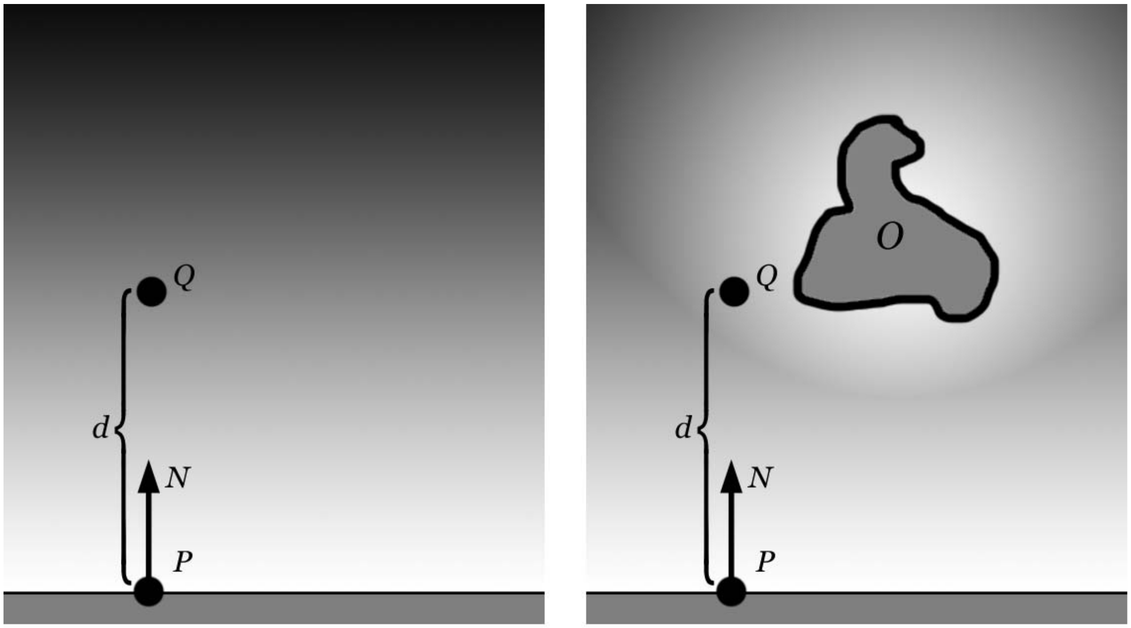


***Figure 6**. (a) As we move away from a point P in the normal direction N, the SDF will be proportional to the distance from P – for example, at point Q - in the absence of any*

nearby 'occluding' objects. (b) Another object causes the SDF to be closer to 0 (white) than expected, at point Q.

Figure 6 shows what happens at increasing distances from the surface of an object, with and without the presence of multiple objects (or oven a single object with a large-scale concave shape). According to the definition of our SDF, at a distance d from P in a direction N perpendicular to the surface at P, we expect the SDF to be:

$$SDF(P+d\,N) = -d$$

This holds only in a scene with no objects 'in front of' P. However, in the presence of another object in the vicinity of the sampled point, the SDF will be closer to 0 (Figure 6b). (The value cannot be more negative, due to the property that at every point, the SDF records the *minimum distance* to the closest surface point).

This property allows us to take measurements of the occlusion / 'openness' of space at increasing distances from P. In particular, if we sample at n points away from P, at distances $d_0, d_1, \ldots d_n$ with $d_0=0$, and $d_i < d_{i+1}$, we expect

$$\sum_{i=0}^{n} SDF(P + d_i N) = -\sum_{i=0}^{n} d_i$$

in the absence of occluders.

Any difference from this represents some degree of occlusion of P along the direction N – and we visualize this difference as before, using an exponential:

$$C = e^{k\sum SDF(P+d_i N)+d_i}$$

where C is the output of the shader.

Each sample (at a distance d_i) can be thought of as capturing the occlusion effects of objects at a distance d_i. To simulate the increased blurring effect of distant shadows, and to avoid artifacts that look like edges in the shadows caused by the point at $P+ d_i\ N$ 'swinging' around rapidly as P and N change, we sample different, pre-filtered copies of the SDF at each sample point. The pre-blur filter kernel size for each sample is set to be proportional to each d_i. That is, we sample a blurred copy of the SDF with the radius of blurring proportional to the distance of the sample from P.

This maps well to the GPU volume texture implementation when the d_i's are spaced according to powers of two – that is, $d_i = 2\ d_{i-1}$ for $i > 1$, with d_1 set to the width of the voxels of the highest resolution volume texture. The pre-filtered volume textures for each SDF are then stored in the mip-map chain for the volume texture, and can be efficiently sampled using mip-level biased texture load instructions in the pixel shader.

Everything discussed so far makes no reference to the orientation of the sky hemisphere itself. As described, the result ‘C’ of the shader given above approximates ambient occlusion, which does not depend on any kind of directional light. To emulate the effect of the sky lighting coming from an oriented, infinitely distant hemisphere, we simply need to skew the sample positions ‘upwards’ towards the sky. This naturally introduces a bias in the sample positions that causes shadows to be cast downwards, and upward facing surfaces to be lit more brightly. We simply ‘bend’ N towards the skylight by replacing N with:

$$N' = N + \alpha U$$

(where U is an upward pointing unit vector giving the orientation of the sky, N is our normal at P and α is a ‘sky weighting factor’ from 0 to 1).

Figure 7 shows the result of rendering a scene with these techniques, and overall quite a convincing skylighting effect is achieved in real time. (cf Figure 1, rendered using Autodesk 3D Studio Max 8’s skylight). Note that there is not even any explicit ‘$N \cdot L$’ term used – the lighting comes purely from the use of the bent normals described above, with a value of α of 0.5. The entire scene is represented by a single 128x128x128 SDF stored in an 8 bits per pixel monochrome volume texture. The shader used to render Figure 7 is given in listing 1. The image in Figure 7 was made with a *static* SDF computed once on the CPU. All that remains is to describe a technique by which we can compute the SDF of an arbitrary scene, at runtime. This will allow the rendering of dynamic scenes.

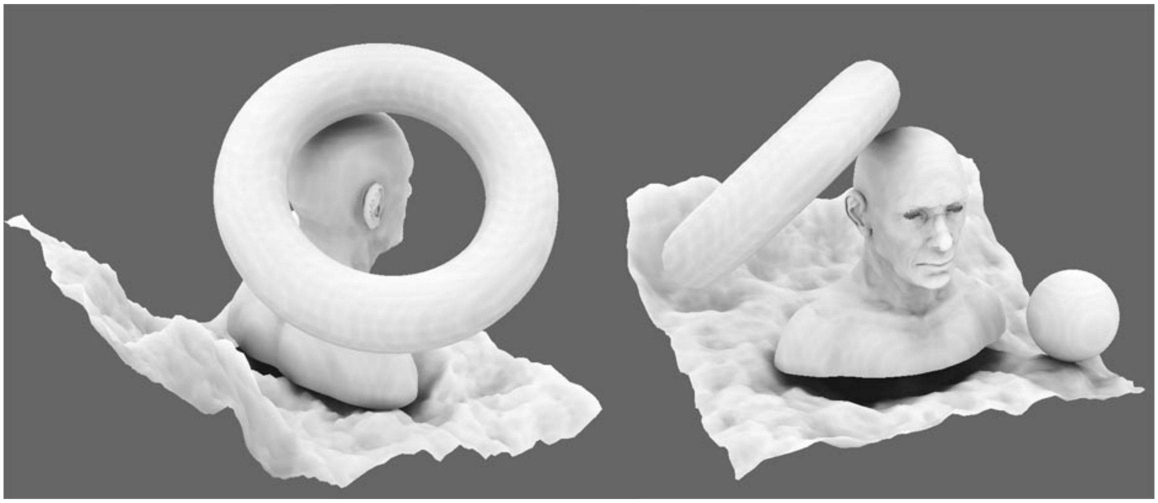


***Figure 7**. A scene rendered using a single, static 128x128x128 SDF sampled 4 times per rendered pixel at 4 different mip levels. All shading comes directly from the SDF technique.*

```
struct VolSampleInfo
{
 float3 minuv;
 float3 maxuv;
 float3 uvscale;
 float4 numtiles; // across in x, down in y, total slices in z,
                   // zfix = 0.5/numtiles.z
};

float4 SampleVolTex( sampler       bigtex,
                     float3        uvw,
                     VolSampleInfo vi,
                     sampler       vtl)
{
 uvw.xy*=vi.uvscale.xy;
 uvw.z-=vi.zfix;
 uvw.xyz=clamp(uvw.xyz,vi.minuv,vi.maxuv);

 float2 uv=uvw.xy;//*0.125;
 float4 fix = tex2D(vtl, uvw.z);
 float4 s1 = tex2D(bigtex,uv+fix.xy);
 float4 s2 = tex2D(bigtex,uv+fix.wz);
 return lerp(s1,s2,frac(uvw.z*vi.numtiles.z));
}

float4 MeshPS(BASIC_OUTPUT i) : COLOR0
{
 float3 n =normalize(i.Normal);
 float3 vec2light(0,0,1);
 float3 pos = i.WorldPos ;
 float3 delta = n* 0.03 + vec2light * 0.03;
 float4 light = exp(
  SampleVolTex(BasicSampler , pos+delta, vi,  VolTexLookupSampler)        +
  SampleVolTex(BasicSampler1, pos+delta*2, vi1, VolTexLookupSampler1)*1.2 +
  SampleVolTex(BasicSampler2, pos+delta*4, vi2, VolTexLookupSampler2)*1.4 +
  SampleVolTex(BasicSampler3, pos+delta*8, vi3, VolTexLookupSampler3)*1.8
 );
 return i.Diffuse * light * GlobalBrightness;
}
```

Listing 1. *DirectX HLSL pixel shader code used to render figure 7. Since current GPUs don't allow rendering directly to the slices of a volume texture, the SampleVolTex function emulates a trilinear sample by sampling a 2D texture on which the slices are laid out in an order determined by a point sampled 1D lookup texture (VolTexLookupSampler in the code)*

9.7 Generating the SDF on the GPU

The final area to describe is how the SDF volume texture is generated in real-time. Many grid based approaches take as their starting point, a volume texture (in 3D) or texture (in 2D) initialized with binary values representing whether the given voxel is inside or outside the objects in the scene (see Figure 3a for an example). We call this the 'binary image'.

The simple brute force approach consists of simply looping over every voxel in the space, and then for each one, searching for the nearest voxel with opposite inside/outside-ness and recording its distance. Figure 3b was created in this manner, and Listing 2 shows C code for computing a 2D SDF on a 256x256 binary bitmap in this manner.

```
void ComputeSDF(unsigned char in[256][256], unsigned char
out[256][256])
{
 for (int y=0;y<256;y++) for (int x=0;x<256;x++)
 {
      int d=W*H;
      int p=in[y][x];
      for (int x2=0;x2<256;x2++) for (int y2=0;y2<256;y2++)
      {
           if (in[y2][x2]!=p)
           {
                int d2=(x-x2)*(x-x2)+(y-y2)*(y-y2);
                if (d2<d) d=d2;
           }
      }
      d=(int)(sqrt(d));
      if (d>127) d=127;
      if (p) d=-d;
      out[y][x]=d+128;
 }
}
```

***Listing 2**. This code was used to generate Figure 3b*

The brute force method however is generally not tractable in 3D, since it is of cost $O(v^2)$ where v is the total number of voxels in the scene. It is certainly not fast enough to use per-frame in an interactive application. Many techniques – such as the Chamfer Distance Algorithm (CDA) have been explored which use small nonlinear kernels to iteratively 'grow' an approximate band of distance values around the boundary of the objects, starting with the binary representation. Each iteration of the filter over the voxel space expands the area which has been correctly initialized by one voxel. See [Danielson80], [Price05], [Grevera04] for examples and pseudo-code.

Figure 5 was rendered using a SDF computed on the CPU using an algorithm similar to the dead reckoning algorithm of [Grevera04].

As mentioned in section 12.5, the SDF need not contain Euclidian distances to achieve an attractive result; indeed, simple Gaussian blurring of the binary image suffices to be able to judge curvature in the sense of Figure 4. Figure 7 was rendered using an SDF generated by repeatedly blurring and downsampling a 512x512x512 binary volume image of the scene, to create a 128x128x128 pseudo-SDF volume texture with 4 mip-maps down to 16x16x16.

9.8 Generating the binary image of a dynamic scene

Whether the SDF is generated through blurring, the Chamfer Distance Algorithm or a full Euclidian distance calculation, some GPU friendly way of generating the SDF volume texture is required. A key observation is that in many cases, a scene will be made up of a number of (possibly moving) objects. Each object need only be able to rapidly compute its own local SDF, before being composited into the final volume texture using the 'min' blending mode of the GPU to generate the 'global' SDF.

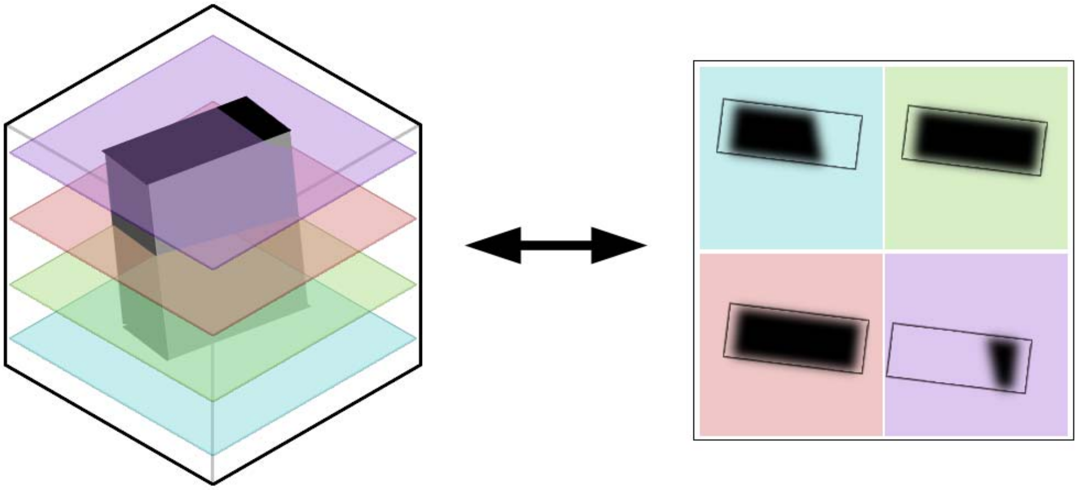


***Figure 8**. a) due to the limitations of current GPUs / APIs which do not allow direct rendering to the voxels of a volume texture, the slices of the SDF volume are laid out on a large 2D texture. The code to sample it is given in Listing 1. (b) a simple vertex shader which intersects the 4 edges of an OBB which most align with the volume slicing axis (normally 'z') allows the intersection of the OBB with the volume slices to be computed efficiently in a single draw call. In the right of the diagram, the vertex shader's output quads are shown outlined; a pixel shader computes the actual SDF at each point within those quads (shown as blurry black regions).*

Since we are rendering into a volume rather than to the usual 2D render target supported by GPUs, we have to lay out the volume in slices (Figure 8). In our implementation, we created a vertex shader that was able in a single D3D draw call, to calculate the intersection of an arbitrary oriented bounding box with all the slices of the volume, and execute a pixel shader to update the SDF for only those 'voxels' which are inside the bounding box (Figure 8b). In this way, the SDF updating algorithm consists of:

```
Clear SDF render target representing the volume V of the entire
scene – assuming a 2D layout of the 3D slices as in Figure 8
For each object O in the scene
      Compute the oriented bounding box (OBB) of O
      Compute the intersection of the OBB of O with the
          slices of V
```

```
For each pixel within the intersection regions,
     Compute the approximate SDF of O,
     ‘min’ blend the result into the SDF render target
```

Depending on the type of object O, we compute its SDF in one of several ways:

1. For a cuboid or ellipsoid, the SDF can be computed analytically.

2. For a rigidly deforming body with a complex shape, the SDF can be precomputed and stored as a volume texture in the local space of O

3. For a ‘height field’ object whose surface can be described as a single valued function of 2 coordinates (x, y), the SDF can be approximated directly from a blurred copy of the height field data describing the surface shape.

4. For a star shaped object (such as that shown in Figure 9) where the surface can be described as a single valued function giving the radius of the object along the direction to the object’s local origin, the SDF can be approximated directly from a blurred copy of the ‘height field’ / Z buffer stored in a cubemap.

Options 1 and 2 are suitable for non-deforming (but affine transformable) objects. Options 3 and 4 (including a variant of 3 in which two height fields are glued back-to-back to create a more general shape) are useful for objects which may deform their shape from frame to frame. In these cases, the height field texture (or cube map for option 4) can be rendered dynamically to an offscreen buffer, using the GPU in a similar manner to the way in which shadow maps are computed on the GPU. The blurring of the Z buffer can also take place efficiently on the GPU; by rendering both the Z value and its square and then blurring both, (as described in [Donnely05] in the context of Variance Shadow Maps) the variance of Z over a range of blurred pixels can be used along with the mean Z to compute the width of the falloff of the SDF along the Z axis. (Figure 9b).

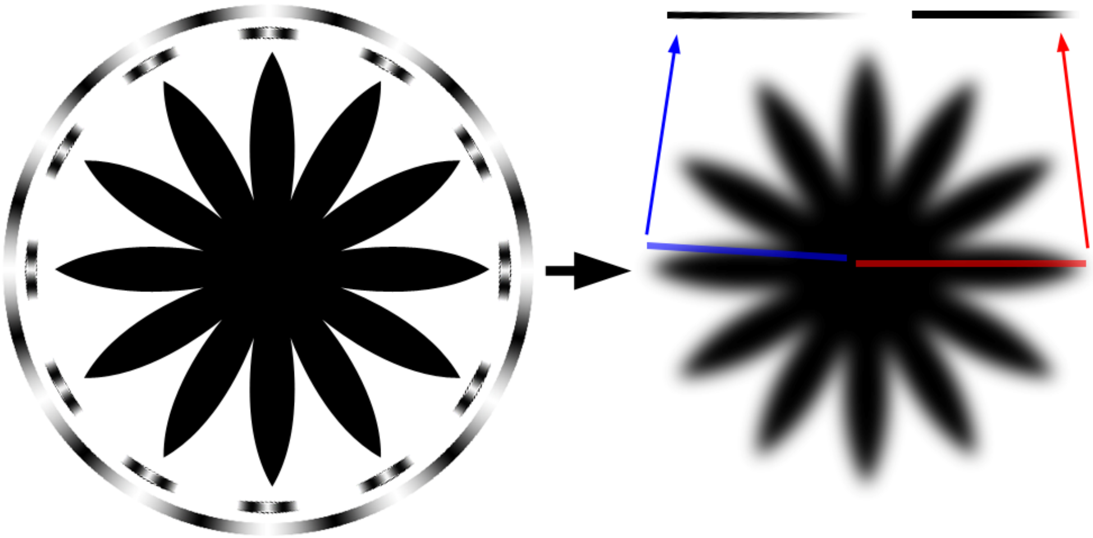


Figure 9. *A star shaped object can have its SDF computed approximately from a radial cube map (or, in 2D as in this figure, a radial 1D texture) which stores the surface Z for*

each theta. This is visualized above by the outer ring in (a). When blurring this Z buffer, the variance in Z is tracked as in [Donnely05] and used to scale the rate of falloff of Z in the SDF, since areas of high Z variance (marked by black parts of the inner ring in (a)) have a softer falloff from black to white along the radial direction, as shown in (b) by comparing the blue radial direction (high variance), with the red radial direction (low variance).

Figure 10 shows a 'Cornell' type box rendered using this algorithm, consisting of 6 cuboids (5 walls and one central cube) whose local SDF are computed analytically in a pixel shader and composited into a 128x128x128 SDF for the whole scene. Figure 11a shows a similar scene with the central box replaced with a complex mesh whose SDF was precomputed in local space (option 2 above); Figure 11b shows a star shaped object with dynamically generated SDF based on a cube map (option 4).

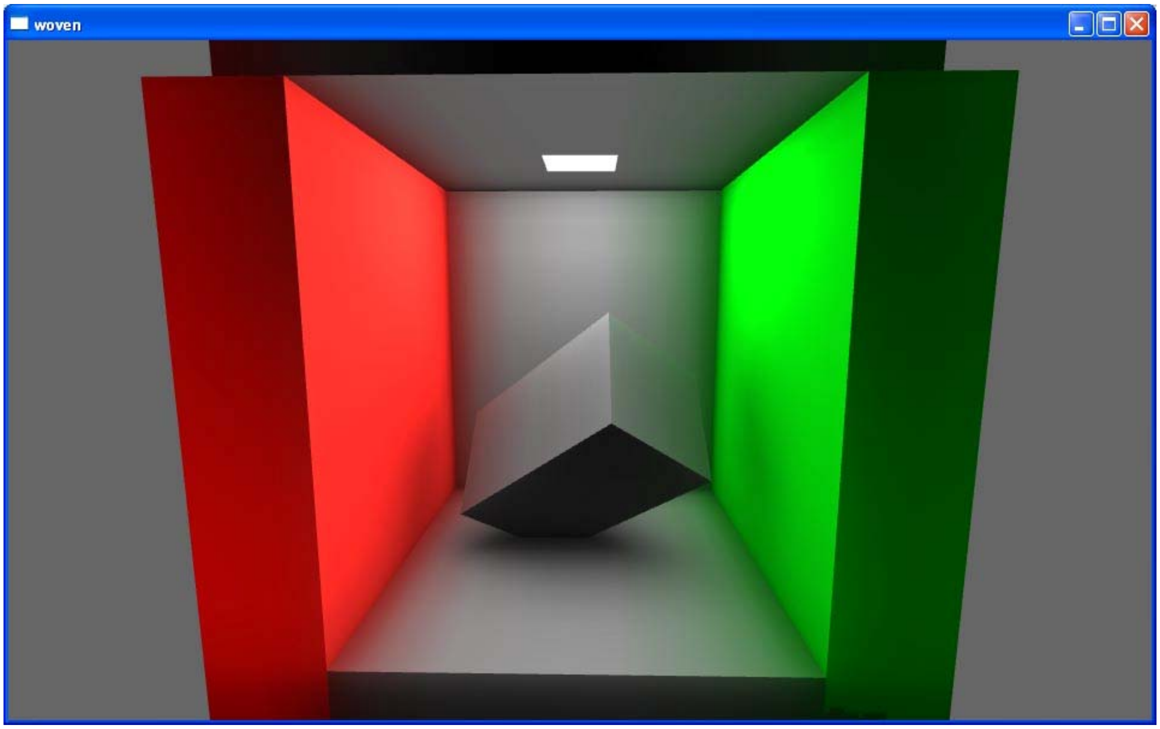


***Figure 10**. An oriented cube in a Cornell box. All shadowing in this image comes from the SDF; in addition surfaces were lit using an N · L term scaled by the results of the SDF lookups.*

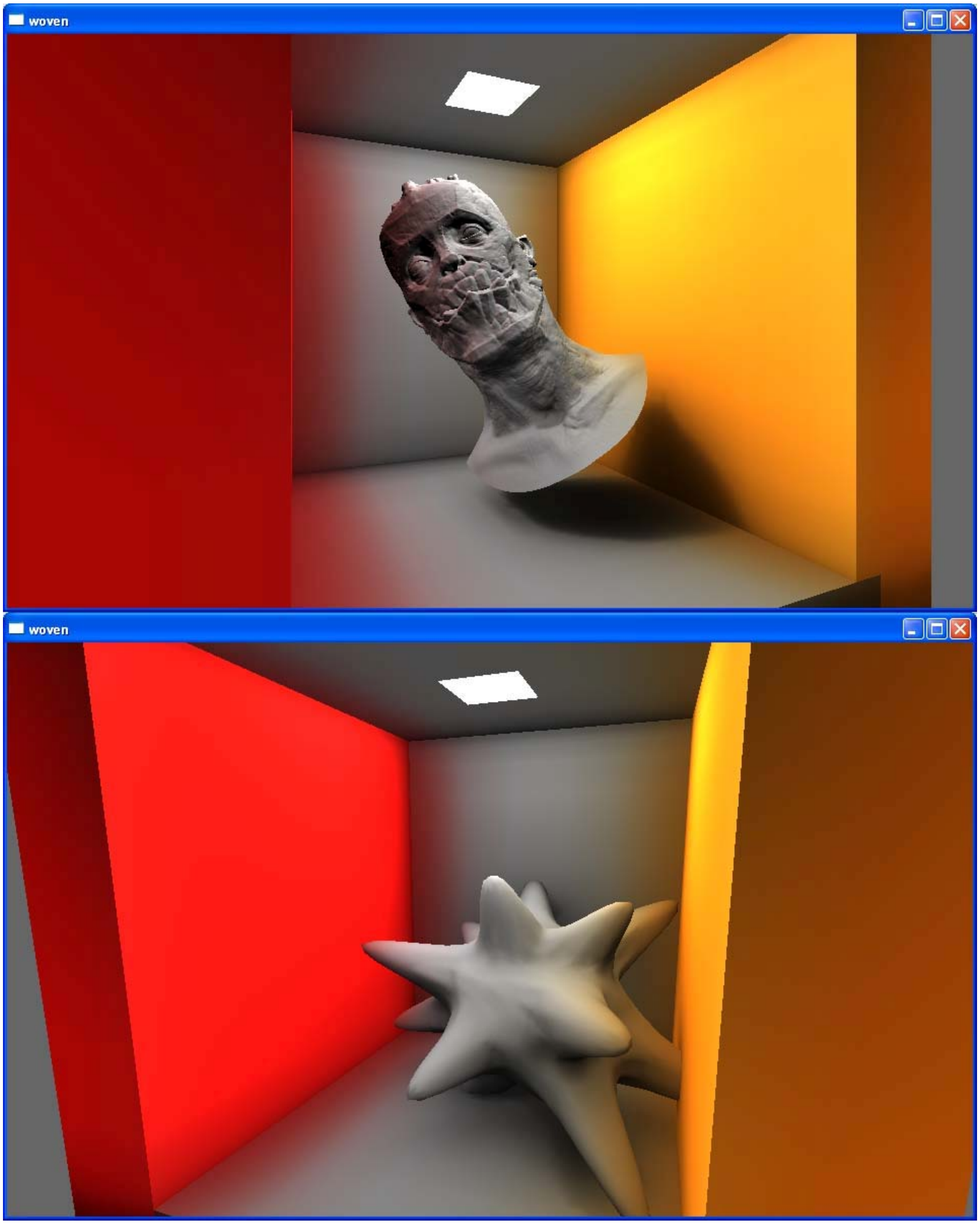


***Figure 11**. Two example scenes with more complex object shapes. (a) has 2 million polygons and a precomputed local SDF stored in a 32x32x32 volume texture. (b) has a dynamically rendered 32x32x6 cube map of radii which is used to approximate its local SDF.*

9.9 Approximate colour bleeding and future directions

Figures 10, 11a and 11b all exhibit small amounts of colour bleed. This was achieved by computing the SDFs separately for red, green and blue channels, and then biasing the SDF in each channel for each object by its diffuse surface colour. The red reflective wall had its red channel of its local SDF increased by a small constant factor, causing all nearby SDF lookups to return a slightly red hue. Despite being physically incorrect, in the sense of not accurately capturing the physics of light bouncing from one diffuse surface to another, it's this kind of simple, free yet aesthetically pleasing trick which can go a long way in a real-time environment.

In the spirit of 'learning from mistakes', the success of this kind of non-physical 'short-cut' also suggests an extension to arbitrary light sources (rather than just sky-lights) based on an interpretation of this skylighting algorithm that has more in common with the Ambient Occlusion Fields of [Kontikanen05] than SDFs. If the volume texture placed over the scene, is considered more as an approximation of the light reaching each point in space, as opposed to the SDF of the occluders in that scene, then an alternative algorithm supporting an arbitrary number of emissive objects in the scene (rather than just a distant skylight). The idea is to start with a blurry copy of the 'binary image' of the scene, stored in a volume texture. In addition, an even lower resolution volume texture is computed containing a vector for each point, containing a direction pointing away from the average of nearby light sources. This volume can be thought of as the first 4 terms (constant and linear) spherical harmonic coefficients of an irradiance volume that does not take into account occluders.

The latter volume, which we shall term 'the light direction' function, can be used to 'advect' the contents of the pseudo-SDF volume texture. This effect is best visualised by observing the music visualisation features of media players such as Apple's iTunes™ or WinAmp. These take a simple bitmap image, and at each frame radially zoom and blur the image, feeding the result back into the next frame. If you imagine radially blurring the SDF volume texture according to the directions stored in the 'light direction' volume texture (in the case of a single point light source, radially from the light), the resulting volume will converge towards a good-looking approximation to the irradiance at every point in space. Figure 12 shows this effect in 2D.

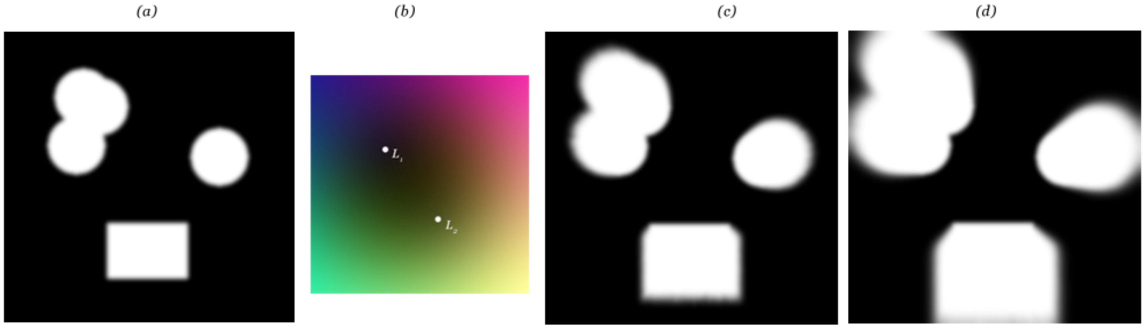


***Figure 12**. (a) starting with a binary image of the scene's occluders ('SDF'), and a lower-resolution texture encoding light flow direction at every point (b), the SDF texture is repeatedly zoomed, blurred and blended with itself (c) to create an approximation to*

the irradiance at every point (d) – shown with colours inverted. This result can then be used to render the scene with soft shadowing from the light sources included in the original light flow texture.

9.10 Conclusion

The flexibility of GPUs was exploited in this description of an unusual soft-shadow rendering algorithm. Although it is of limited use since it requires the whole scene to be represented in a volume texture, it demonstrates a process of art-led discovery, approximation and exploration in real-time graphics rendering which the author believes will continue to be one of the driving forces behind exciting, visually unique games and real-time applications, running on commodity GPU hardware.

9.11 Bibliography

Danielson, P.-E., 1980. Euclidian Distance Mapping. *Computer Graphics and Image Processing* 14, pp. 227-248

Donnelly, W. 2005. Per-Pixel Displacement Mapping with Distance Functions, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Chapter 8, Matt Pharr (ed.), Addison-Wesley. http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch08.pdf

Donnelly, W., Lauritzen, A. 2006. Variance Shadow Maps. In the proceedings of *ACM SIGGRAPH 2006 Symposium on Interactive 3D Graphics and Games*.

Evans, A. 2005. Making Pretty Pictures with D3D. GDC Direct3D Tutorial 2005, http://www.ati.com/developer/gdc/D3DTutorial07_AlexEvans_Final.pdf

Everitt, C. 2001. Interactive Order-Independent Transparency. NVIDIA white paper, 2001, http://developer.nvidia.com/object/Interactive_Order_Transparency.html

Frisken, S.F.; Perry, R.N. 2002. Efficient Estimation of 3D Euclidean Distance Fields from 2D Range Images, *Volume Visualization Symposia* (VolVis), pp. 81-88

Frisken, S. F., 2006. Saffron: High Quality Scalable Type for Digital Displays, Mitsubishi Electric Research Laboratory (MERL), http://www.merl.com/projects/ADF-Saffron/

Grevera, G. J. 2004. The ‘‘Dead Reckoning’’ Signed Distance Transform, *Computer Vision and Image Understanding* 95 (2004) pp. 317–333.

Kajiya, J. T. 1986. The Rendering Equation. *Computer Graphics* 20 (4), 143-149

KONTKANEN, J., LAINE, S. 2005. Ambient Occlusion Fields, In the proceedings of *ACM SIGGRAPH Interactive Symposium on 3D Graphics and Games*

MASUDA, T. 2003. Surface Curvature Estimation from the Signed Distance Field. 3dim, p. 361, *Fourth International Conference on 3-D Digital Imaging and Modeling* (3DIM '03)

OAT, C. 2006. Ambient Aperture Lighting, "Advanced Real-Time Rendering in 3D Graphics and Games", Course 26, ACM SIGGRAPH

POPE, J., FRISKEN, S. F., PERRY, R.N. 2001. Dynamic Meshing Using Adaptively Sampled Distance Fields, Mitsubishi Electric Research Laboratory (MERL) Technical Report 2001-TR2001-13

PRICE, K. 2005. Computer Vision Bibliography Webpage. http://iris.usc.edu/Vision-Notes/bibliography/twod298.html