APPROVAL SHEET

Title of Thesis: Real-time Soft Shadows on the GPU via Monte Carlo Sampling

Name of Candidate: Aaron Curtis
Master of Science, 2009

Thesis and Abstract Approved: ________________________
Dr. Marc Olano
Associate Professor
Department of Computer Science and
Electrical Engineering

Date Approved: ______________________________

Curriculum Vitae

Name: Aaron Curtis

Permanent Address: 9200 Livery Lane, Apt B; Laurel, MD 20723

Degree and date to be conferred: Master of Science, May 2009

Date of Birth: October 14, 1980

Place of Birth: Baltimore, MD

Secondary Education: Hereford High School, Parkton, MD

Collegiate institutions attended:

University of Maryland Baltimore County, M.S. Computer Science, 2009

University of Maryland Baltimore County, B.S. Computer Science, 2006
Major: Computer Science
Minor: Mathematics

University of Maryland Baltimore County, B.S. Mechanical Engineering, 2002
Major: Mechanical Engineering

Professional publications:

Curtis, Aaron and Olano, Marc. 2009. Real-time Soft Shadows on the GPU via Monte Carlo Sampling. *High Performance Graphics*. (submitted)
[Possible stuff about the High Performance Graphics paper] (to appear)

Professional positions held:

Teaching Assistant. CSEE Department, UMBC. (September 2007 – June 2009).

Assistant Software Engineer. RWD Technologies. Baltimore, MD. (December 2006 – August 2007).

Intern. RWD Technologies. Baltimore, MD. (September 2005 – August 2006).

ABSTRACT

Title of Thesis: Real-time Soft Shadows on the GPU via Monte Carlo Sampling

Aaron Curtis, Master of Science, 2009

Thesis directed by: Dr. Marc Olano, Associate Professor
Department of Computer Science and
Electrical Engineering

Realistic shadows present a difficult problem in real-time rendering. While techniques for rendering hard edged shadows from point light sources are well established, attempts to incorporate soft shadows typically suffer from inaccuracies or poor performance.

Our algorithm makes use of recent advances in GPU randomization to perform Monte Carlo sampling of points on an area light source. Rays are then traced to the sampled points, using the shadow map as a discretized representation of occluders in the scene. The accuracy of this method can be improved through the use of multiple shadow maps, which together are able to better approximate the scene geometry.

As with conventional shadow mapping, our method is performed entirely on the GPU, does not require any precomputation, and can handle fully dynamic scenes with arbitrary geometric complexity. The quality of the generated shadows is comparable to that of offline rendering algorithms such as ray tracing, while performance remains real-time, on par with existing techniques.

Real-time Soft Shadows on the GPU via Monte Carlo Sampling

by

Aaron Curtis

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2009

Ⓒ Copyright Aaron Curtis 2009

TABLE OF CONTENTS

LIST OF FIGURES . . . . **iv**

LIST OF TABLES . . . . **vii**

Chapter 1 INTRODUCTION **1**

Chapter 2 RELATED WORK AND BACKGROUND MATERIAL **3**

2.1 Real-time Soft Shadowing 3
2.2 Height Field Intersection 6
2.3 Review of Shadow Mapping 7
2.4 Graphics Hardware 8

Chapter 3 APPROACH **12**

3.1 Monte Carlo Sampling 16
3.2 The Inadequacy of a Single Depth Map 18
3.3 Ray Tracing with Two Depth Layers 19
3.4 Additional Depth Layers 21
3.5 Hierarchical Search 23
3.6 Penumbra Classification 25
3.7 Other Acceleration Techniques 26

Chapter 4 RESULTS **30**

4.1 Limitations 31

4.2 Performance 32

4.3 Impact of the MD5 Hash 35

Chapter 5 CONCLUSION **40**

5.1 Future Work 40

Appendix A REDUCED MD5 HASH IMPLEMENTATION **42**

REFERENCES **46**

LIST OF FIGURES

1.1 Sample results for our shadowing technique. Scenes were rendered at 20 and 30 frames per second, using a 512x512 texel shadow map and nine rays per fragment. 2

2.1 The graphics pipeline. Vertex data streams in from the CPU and is converted to pixels on the screen in a series of stages. On current hardware, each of the programmable stages can access texture memory on the GPU. . 9

2.2 The GeForce 8800 architecture. Source: Nvidia. The vertex, geometry, and fragment stages are handled by unified stream processors. 10

3.1 An area light source and corresponding shadows. In our approach, the depth map is rendered from a projection point placed behind the center of the light. 13

3.2 Progression from probabilistic shadowing to true soft shadows. Images (b)–(d) show the effect of tracing multiple rays per fragment while (e) adds a filtering pass to (d); (a) and (e) show the same scene. 15

3.3 Effects of different sampling patterns. Images were rendered using two depth layers and hierarchical search, with nine rays per fragment. 17

3.4 Geometry reconstruction using two depth layers. Possible occluders are bounded between the near layer (red) and the far layer (blue). It is possible to use a third depth layer (not shown) to verify the existence of occluders inside the bounded volume. 20

3.5 Self Shadowing. A third depth layer (b) allows for correct softening of the shadow cast by the dragon's horn. Images were generated using 4 rays per fragment; (a) uses hierarchical search while (b) uses linear/binary. 21

3.6 A four-level N-buffer. A texel at level n stores the maximum luminosity of the surrounding texels in a neighborhood of size 2^n at level 0. The texels highlighted in black have neighborhoods highlighted in red, and were calculated by taking the maximum of the four texels from the previous level, at the locations marked with red Xs. 24

3.7 Determination of the portion of the depth map containing potential occluders. All possible rays from a point in the scene intersect a plane at the minimum occluder depth. The intersection region projects onto a subset of the depth map. 26

3.8 Classification of fragments. Green areas are potentially in penumbra, while full calculations are only performed in blue areas. 27

3.9 Ray shortening. Ray endpoints are modified based on the bounds for possible occluder depths, resulting in a smaller affected region of the depth map. 28

4.1 Shadowing of the dragon model. Labels show the search type and number of rays per fragment. Shadow map size is 512x512. 37

4.2 Shadowing of an elongated cube, a difficult case for many soft shadowing algorithms. Labels show the search type and number of rays per fragment. Shadow map size is 512x512. 38

4.3 Shadow map resolution scaling for hierarchical search with deferred shading enabled. Labels show the number of rays per fragment and which scene was used. 39

4.4 Screen resolution scaling for hierarchical search with deferred shading enabled. Labels show the number of rays per fragment and which scene was used. 39

LIST OF TABLES

4.1 Performance scaling with shadow map resolution for the dragon scene (Figure 4.1). Screen resolution is 1024x768. Values are in frames per second. . 33

4.2 Performance scaling with shadow map resolution for the elongated cube scene (Figure 4.2). Screen resolution is 1024x768. Values are in frames per second. 33

4.3 Performance scaling with screen resolution for the dragon scene (Figure 4.1). Shadow map resolution is 512x512. Values are in frames per second. 34

4.4 Performance scaling with screen resolution for the cube scene (Figure 4.2). Shadow map resolution is 512x512. Values are in frames per second. 34

Chapter 1

INTRODUCTION

Shadows are an essential component of computer-generated images, allowing for easier recognition of spatial relationships as well as offering aesthetic appeal and increased realism. However, real-time rendering has until recently been limited to hard shadows, based on the assumption that light sources are single points. In reality, light sources occupy an area in space, leading to soft shadows with smooth transitions between fully lit and fully shadowed points. Rendering such regions of penumbra requires the calculation of the fraction of light received by each point in the scene.

Shadow mapping (Williams 1978) has become the de facto standard for rendering shadows in real-time applications, given its relative simplicity and its ability to scale well with increasing scene complexity. Additionally, it requires no precomputation and can handle fully dynamic scenes as well as any type of rasterizable geometry. We therefore introduce a new technique that extends shadow mapping to handle area light sources, while preserving most of the algorithm's advantages.

A recent class of soft shadow algorithms (Guennebaud, Barthe, & Paulin 2006; Schwarz & Stamminger 2007; Atty *et al.* 2006) has been developed based on the assumption that the depth map employed in traditional shadow mapping can be used as a discretized representation of all the potential occluders in a scene. We make this same assumption, and use the depth map to trace rays through the scene, from potentially shadowed points to points on the surface of an area light. To properly estimate the fraction of illumination received by each point, we rely on recent advances in GPU randomization (Tzeng & Wei 2008) to perform Monte Carlo sampling of points on the surface of the light, then combine the results of several samples.

The use of a depth map as a representation of the scene geometry can lead to innaccurate shadows in the final rendering. The depth map, being a two-dimensional array of depth values, is unable to represent overlapping objects or to distinguish between thin, planar objects and ones that extend deep into the scene. We therefore show how our method can make use of multiple depth maps to improve the shadow quality.

The primary benefit of our technique is the creation of physically accurate soft shadows, resulting in images with a high visual quality. Figure 1.1 shows some sample results. Performance is comparable to existing state of the art algorithms.

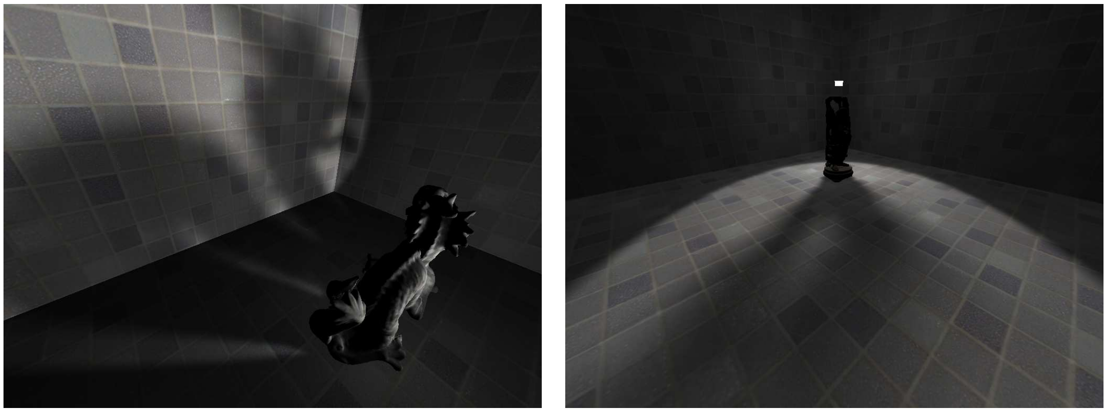


FIG. 1.1. Sample results for our shadowing technique. Scenes were rendered at 20 and 30 frames per second, using a 512x512 texel shadow map and nine rays per fragment.

Chapter 2

RELATED WORK AND BACKGROUND MATERIAL

Simple and effective methods for handling area light sources have long existed in offline rendering. We refer the reader to the survey by Woo et al. (1990) for an exhaustive list. Ray tracing in particular offers an elegant solution to shadow generation; Cook et al. (1984) describe modifications to the basic raytracing algorithm that allow for “fuzzy” effects, of which soft shadows are one of the simplest. Similarly, path tracing as described by Kajiya (1986) allows for soft shadowing by replacing ray tracing’s branching ray tree with a single probabilistically determined path. At its heart, our work relies on these same basic concepts, stochastically tracing rays through the scene to determine light source visibility.

2.1 Real-time Soft Shadowing

A more recent survey (Hasenfratz *et al.* 2003) catalogs many of the earliest attempts to render soft shadows in real time, most of which build upon the two major real-time hard shadowing techniques—shadow mapping (Williams 1978) and shadow volumes (Crow 1977). The latter was most notably extended with penumbra wedges (Akenine-Möller & Assarsson 2002), which add additional geometric primitives at silhouette edges in object space. This yields relatively accurate shadows as long as object silhouettes do not overlap. Some work has been done to address the problem of overlaps (Forest, Barthe, & Paulin

2006), but the method also inherits the general disadvantages of shadow volumes, most importantly that it scales poorly with scene complexity.

Other methods fare better with respect to scene complexity, but are limited in the types of scenes they can support, or eschew physical correctness in favor of rough approximations. Several techniques concentrate only on extending penumbrae out from a traditional shadow map (Chan & Durand 2003; Wyman & Hansen 2003), while true penumbrae extend both inward and outward. Brabec and Seidel (2002) search a shadow map and modulate the illumination based on estimated occluder locations. Soler and Sillion (1998) convolve images of occluders with the light to determine shadowing, but this is only accurate for planar objects and requires that occluders and receivers be disjoint. Still other researchers (Agrawala *et al.* 2000; Heidrich, Brabec, & Seidel 2000) rely on rendering very many depth maps from multiple points on the surface of a light and combining the results, the overhead of which limits them to static scenes. Interestingly, in the same work Agrawala et al. suggest tracing rays through depth maps to sample light source visibility, though they do not present this as a real-time method. Our technique is similar, but we do not render multiple depth maps from the surface of the light, as projections from viewpoints so near to each other can only provide minimal extra information about occluders in the scene.

A more recent class of algorithms generates penumbrae by blurring the shadow map using a variable-width filter, with the filter width determined by the average occluder depth in the affected region. These algorithms differ primarily in the filtering methods they employ, since conventional filtering cannot be applied to shadow maps. Percentage closer soft shadows (Fernando 2005) rely on percentage closer filtering (PCF) (Reeves, Salesin, & Cook 1987); Annen et al. (2008) rely on convolution shadow maps (CSMs) (Annen *et al.* 2007); and Lauritzen (2007) makes use of variance shadow maps (VSMs) (Donnelly & Lauritzen 2006). All three can achieve aesthetically pleasing shadows in most scenes, although they are only physically accurate for planar occluders, similarly to Soler and Sillion's (1998) work. These algorithms are also limited by the fact that accurately determining the average occluder depth can be computationally very expensive, although Annen et al. note that this can be more efficiently expressed as a convolution. Furthermore, all of the filtering techniques come with certain drawbacks, be it the inability to support pre-filtering in the case of PCF, light leaking in case of VSMs, or high memory use with CSMs.

Beginning with the work of Atty et al. (2006) and Guennebaud et al. (2006) (developed independently), another class of algorithms has been developed based on the assumption that a depth map can be used to represent occluders in the scene as a set of micro-patches. The micro-patches are back-projected into world space and then onto the surface of an area light in order to measure the fraction of light occluded at any point in the scene. This allows for physically correct shadow calculations, so long as the depth map is able to adequately represent all the occluders in the scene. Problems can arise in the projection process when either the patches overlap each other or gaps form between them, leading to over-darkening or light leaks. Guennebaud et al. explicitly address gaps in their algorithm at the expense of more severe over-darkening from overlaps. Bitmask soft shadows (Schwarz & Stamminger 2007) eliminate the problems with overlaps, but come with increased computational expense. Performance in general with backprojection is relatively poor, however these methods can make use of mipmap-like acceleration structures to eliminate large regions of potential occluders and achieve real-time frame rates.

As with backprojection, we also treat the depth map as a discretized set of occluders, but we choose to use this idea to cast rays through the depth map instead. Ray casting in this manner naturally does not suffer from gaps or overlaps and can still benefit from the same acceleration structures.

2.2 Height Field Intersection

Tracing rays through a depth map (or height field) is not a new idea and has recently made appearances in numerous real-time rendering techniques outside the realm of shadowing, including relief mapping (Policarpo, Oliveira, & Comba 2005), refraction rendering (Wyman 2005), and caustics mapping (Shah & Konttinen 2007), among others. A full survey is outside the scope of this paper.

Accurately intersecting a height field is not without its difficulties. The typical method, as described by Policarpo et al. (2005), uses linear search with some fixed step size, followed by binary search when the algorithm suspects the ray has crossed a boundary. As many authors have noted, the linear stage is prone to aliasing when the scene contains fine structures, due to undersampling of the depth map. Calculating a completely accurate intersection would require taking a sample each time the ray crosses a texel boundary, but this is impractical with linear search because of the number of samples that would be required.

In relief mapping, a secondary ray is cast from the intersection point to determine shadowing. The shadow ray uses linear search only, as finding the exact intersection point is unnecessary. In this respect, shadowing is a somewhat unique application of height field intersection; in most other domains, the goal is to find an exact intersection point. Consequently, several intersection algorithms have been developed with faster convergence, such as the secant method used in interval mapping (Risser, Shah, & Pattanaik 2005) or Shah and Konttiten's (2007) method based on Newton-Raphson iterations. These methods are inappropriate for our purposes however, as they assume that an intersection always occurs and that the depth values represent a continuous function.

Other authors suggest the use of pre-computed acceleration structures such as cone step mapping (Dummer 2006) or relaxed cone step mapping (Policarpo & Oliveira 2007), but we require that any such structure be possible to compute in real-time, since the shadow map must be updated every frame. Pyramidal displacement maps (Oh, Ki, & Lee 2006) and Maximum mipmaps (Tevs, Ihrke, & Seidel 2008) offer such an approach, and are both very similar to the hierarchical shadow maps used by Guennebaud et al. (2006). These are constructed similarly to mipmaps, except that values in the coarser levels are based on the maximum values in the finer levels, rather than the average. N-buffers (Décoret 2005) and multi-scale shadow maps (Schwarz & Stamminger 2007) represent another variant of the same idea, allowing for better acceleration while being somewhat more expensive to generate. Although these can be thought of primarily as acceleration structures, they also eliminate the aliasing problems associated with linear search.

As previously mentioned, our algorithm can use multiple depth maps to better approximate the scene geometry. Several existing height field intersection algorithms have been adapted for use with multiple layers, for similar reasons. Policarpo and Oliveira (2006) extend relief mapping to four layers, while Chun et al. (2008) extend pyramidal displacement maps to two.

2.3 Review of Shadow Mapping

Since our method is based on shadow mapping, we present a brief review of the algorithm. Shadow mapping operates in two passes; in the first, the scene is rendered from the point of view of the light source. Instead of rendering color values to the screen, the first pass stores the depth of each pixel in a buffer (or *map*). In the second pass, the scene is rendered normally, from the point of view of the eye. Each pixel is classified as shadowed or unshadowed based on the following procedure:

- Transform the pixel's 3D coordinates from eye-space into light-space.
- Look up the depth stored in the first pass, using the light-space coordinates.

- If the pixel's depth is greater than the stored depth, some other object is occluding the light's view of it, so the pixel is shadowed.

2.4 Graphics Hardware

Real-time rendering effectively requires some form of hardware acceleration; therefore, we present a brief overview of modern graphics hardware. To begin, any rendering system requires some method of representing the objects to be displayed, and while there are a number of ways to describe objects in three-dimensional space, real-time applications predominantly rely on objects tessellated into meshes of vertices connected by triangles. Triangle meshes make efficient use of memory and lead to a relatively simple display implementation involving matrix operations. Consequently, graphics hardware has evolved largely around this approach, and modern systems are capable of handling meshes with hundreds of thousands, or even millions of triangles.

Figure 2.1 shows a traditional graphics pipeline, composed of several stages. In the initial (vertex) stage, vertex data is streamed in from main memory, and may include spatial coordinates, normal vectors, texture coordinates, colors, or other application-specific information. Next, in the geometry stage, triangles are assembled from the vertex data. The following stage, rasterization, converts the triangles from three-dimensional objects into pixels on the screen. The fragment stage then determines the color of each pixel (pixels in this stage are referred to as fragments, since they do not yet fully represent what will appear on the screen; for example, some may be discared due to overlapping geometry). In most cases, the calculations for fragment color are based on interpolated data from the vertex stage. The final stage, display, applies blending and/or anti-aliasing to the fragments and stores them in memory for display on the screen. It is also possible for the final stage to output to a buffer instead, as is the case with the first pass in shadow mapping.

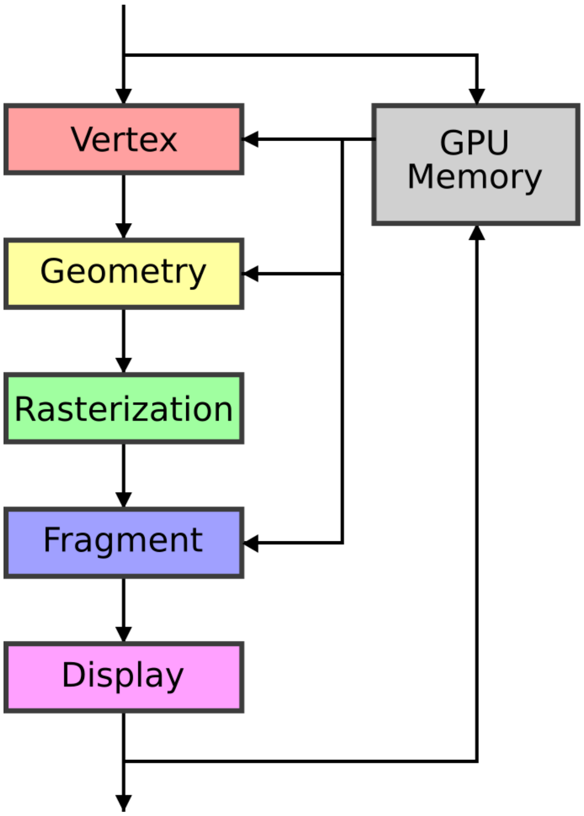


FIG. 2.1. The graphics pipeline. Vertex data streams in from the CPU and is converted to pixels on the screen in a series of stages. On current hardware, each of the programmable stages can access texture memory on the GPU.

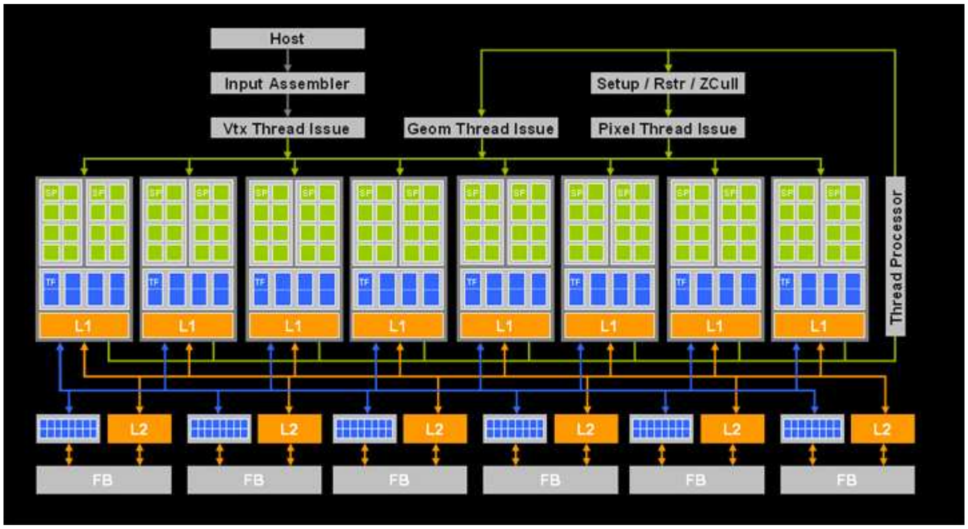


FIG. 2.2. The GeForce 8800 architecture. Source: Nvidia. The vertex, geometry, and fragment stages are handled by unified stream processors.

This model achieves good performance through massive parallelism. For example, in the vertex stage, no vertex depends on any other, so each may be processed independently; the same is true for triangles in the geometry stage and fragments in the fragment stage.

Traditional graphics processing units (GPUs) contain several discrete hardware units with fixed functionality for each stage. However, more recent GPUs contain programmable *stream processors* that are capable of performing the processing for different stages. While some fixed functionality remains, current hardware exposes the vertex, geometry, and fragment stages to programmers. Figure 2.2 shows a block diagram of the Nvidia GeForce 8800 architecture (NVIDIA 2006), which is the hardware we used for testing. Each green block represents a processor; clusters of processors are scheduled to process sets of vertices, triangles, or fragments as necessary. Other vendors, such as ATI, offer comparable hardware.

Our shadowing technique operates primarily in the fragment stage; consequently we have implemented it as a fragment program (or *shader*) for the GPU. While early programmable GPUs required that programs be written in assembly, several high-level languages currently exist that allow access to GPU hardware, with the two primary options being Microsoft’s HLSL (part of DirectX) or GLSL, which is part of the OpenGL standard. The features of the two differ only very slightly, but we have chosen GLSL because of its portability.

The stream processors are in many ways similar to traditional CPUs, so the high level shading languages were modeled on the C family of languages and support most of the familiar control structures. Key differences include a focus on four-component vector data types, a lack of support for pointers, and specialized instructions for accessing texture memory. Support for integer data types has only recently become available, and we make use of it in our implementation.

Chapter 3

APPROACH

We begin with a description of the algorithm in its simplest form, using a single depth map, which is created in the same manner as in conventional shadow mapping, i.e. by rendering the scene from the point of view of the light source and storing depths. However, it is important to note that with an area light, the projection point must be moved back a sufficient distance such that the entire light fits within the view frustum. This distance, $light\ depth$, is given by Equation 3.1, where $light\ size$ is the length of one side of a square light. We restrict the discussion to square lights for simplicity's sake, though the algorithm is easily adaptable to rectangular or circular lights as well.

$$\textit{light depth} = \frac{\textit{light size}}{2 * tan(\textit{fov}/2)} \tag{3.1}$$

Figure 3.1 illustrates the way we set up the light source, along with the corresponding idealized umbra and penumbra regions resulting from a single occluder. In the region labeled *umbra*, any ray drawn from the receiving surface to a point on the light source will intersect the occluder; likewise, in the region labeled *fully lit*, no such rays can intersect the occluder, and in the *penumbra*, rays may or may not intersect the occluder.

We then run a second rendering pass from the point of view of the eye. For each fragment, we trace a ray from a randomly chosen point on the surface of the light to the

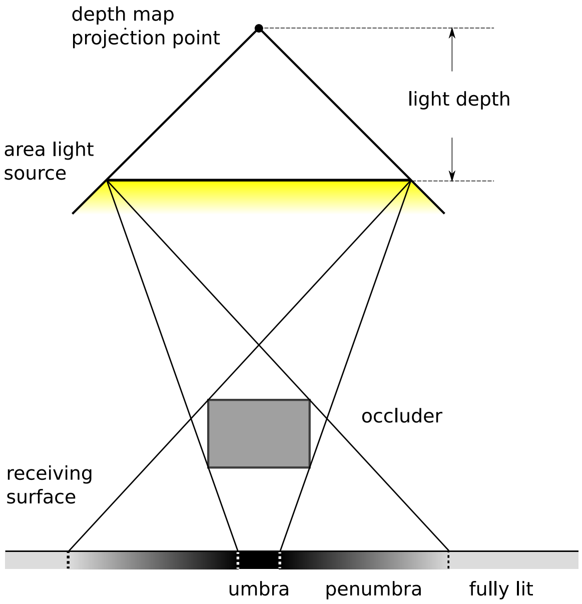


FIG. 3.1. An area light source and corresponding shadows. In our approach, the depth map is rendered from a projection point placed behind the center of the light.

fragment (or the reverse direction works just as well), treating the depth map as though it were de-projected into world space to form a set of micro-occluders that the ray might intersect. If at any point, the ray passes behind one of these micro-patches (that is, the ray is farther from the light), then the fragment is considered shadowed. An implicit assumption is that the patches represent the front surfaces of solid objects extending further into the scene; hence any ray that passes behind one of the micro-patches must strike an object.

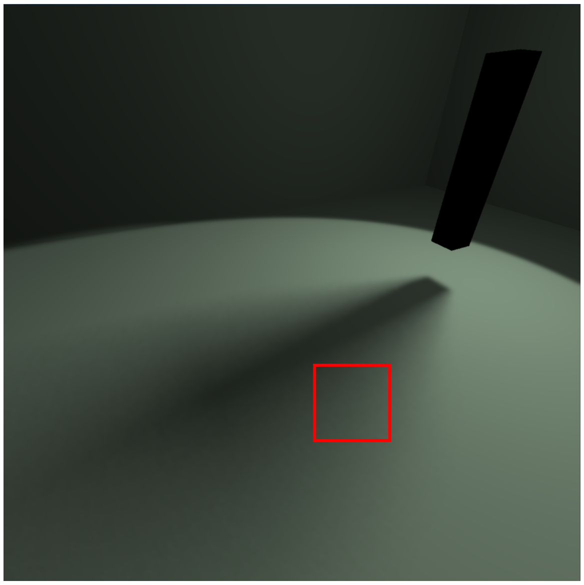
In practice, the ray tracing is accomplished by projecting the ray onto the depth map and sampling points along it linearly, or using hierarchical search methods which we describe in Section 3.5. Placing the projection point behind the area light as described previously ensures that every point along the ray projects to a point in the interior of the depth map; that is, the depth comparison is valid at every point.

Like conventional shadow mapping, our method can suffer from “shadow acne” when rays are cast to unshadowed surfaces, since the ray endpoints have depths exactly equal to those stored in the depth map. To prevent these artifacts, we shorten the rays by a small bias value. A closely related problem occurs when the surface depth varies within the area covered by a single depth map texel—some surface points will inevitably have depths greater than those stored in the depth map, much more so if the surface is at an oblique angle to the light. Our implementation uses an adaptive biasing formula, given by Equation 3.2, in which z is the fragment depth (proportional to the depth texel size in world space), w is the depth map size in texels, $\hat{n}$ is the surface normal, and $\hat{l}$ is the direction of the light. As a practical matter, $\hat{n} \cdot \hat{l}$ must be limited to some small, non-zero value.

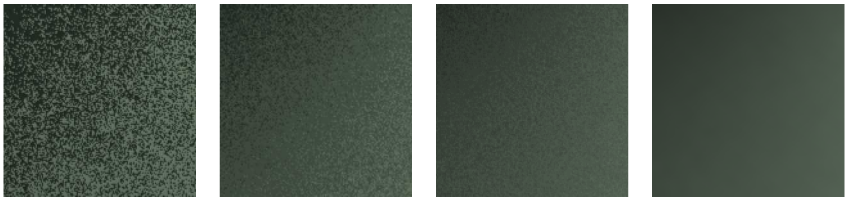
$$\textit{bias} = \frac{\sqrt{2} * z * tan(\textit{fov}/2)}{w * (\hat{n} \cdot \hat{l})} \tag{3.2}$$

In the images that result from our method, points are either fully lit or fully shadowed, but with the probability of being shadowed based on the fraction of the light source that is occluded. We use a combination of techniques to convert this into an image in which points may be partially shadowed; the first is simply to cast multiple rays for each fragment and average the results. Casting multiple rays quickly becomes computationally expensive, so we also perform a Gaussian blur of the shadows in screen space. Figure 3.2 shows the progression from an unblurred, single-ray image to our final result. Note that as the number of rays per fragment is increased, the images begin to approach a soft shadow in appearance, finally reaching the desired result in the last image when the filter is added.

To perform the filtering, it is necessary to render the fully lit scene and the shadow values into separate buffers, perform the blur on the shadows (in two passes using a separable filter), and then recombine the two buffers into a final image. A conventional Gaussian filter would cause unacceptable artifacts by blurring shadows across object boundaries in the scene; therefore, in the second pass we also render depth values in eye space and introduce a *depth sensitive* filter, in which the contribution of each sample is inversely proportional to



(a) penumbra from a large area light



(b) 1 ray (c) 4 rays (d) 9 rays (e) filtered

FIG. 3.2. Progression from probabilistic shadowing to true soft shadows. Images (b)–(d) show the effect of tracing multiple rays per fragment while (e) adds a filtering pass to (d); (a) and (e) show the same scene.

the difference between its depth and that of the central sample (plus 1 to prevent division by zero). A robust implementation would also include normal sensitivity to prevent shadows from blurring across sharp object edges, but such artifacts were generally not visible in our test scenes.

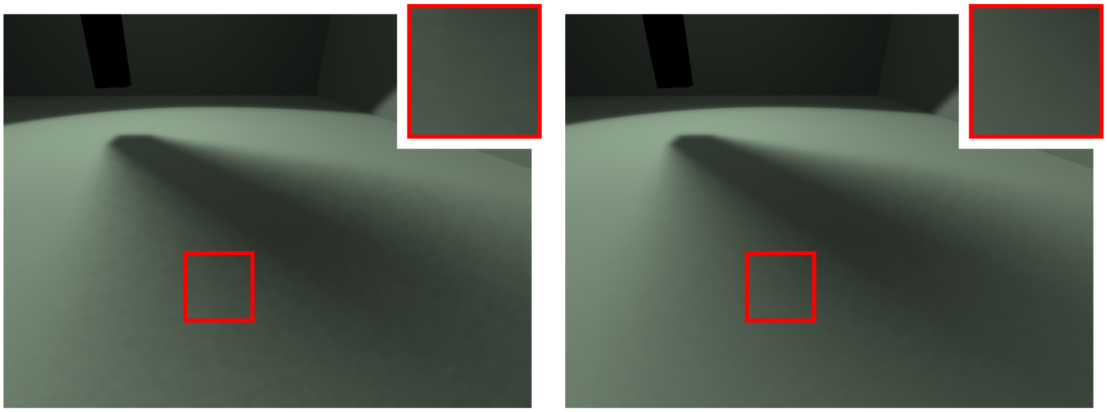
3.1 Monte Carlo Sampling

To obtain sample points on the surface of the light, we require some form of pseudo-random number generator. However, traditional random number generators are unsuited to the massively parallel architecture of the GPU, since they operate sequentially; that is, each call to a traditional generator updates some internal state that is used to create the next random number. Hence each returned value depends on the previous one, beginning with some initial seed. This means that generating n random numbers must take $O(n)$ time, regardless of any parallelism available in the hardware.

To avoid this problem, we make use of an idea introduced by Tzeng and Wei (2008), which is that random numbers can be generated in parallel using a cryptographic hash. We use the screen coordinates of each fragment as the input to the has, and because cryptographic hashes have the property that minor changes to their input (such as moving by one pixel on the screen) produce dramatic changes in the output, the values generated in our algorithm are effectively random. Additionally, each hash can be computed independently, allowing for good performance on the GPU. The specific hash that we use is a variant of MD5; the standard MD5 performs 64 “rounds” on its input, while our version is reduced to 16, allowing for an efficient implementation using a small amount of shader code. For reference, our implementation is included in Appendix A. We experimented with versions using even fewer rounds, but these resulted in visible patterns in our output.

It is unlikely that our reduced MD5 would be remain suitable for cryptographic applications; we merely claim that it is good enough for our purposes. However, we have verified the effectiveness of the hash using the DIEHARD (Marsaglia 1995) test suite, which is the de facto standard for testing random number generators. The suite consists of 15 tests, each of which outputs a varying number of p-values (D’Agostino & Stephens 1986), which should be between 0.01 and 0.99 for a successful run (whether a test passes or not is actually somewhat ambiguous, with the only sure failure being when all p-values are exactly zero or one, but we use the previously given range for consistentency with Tzeng and Wei (2008)). The reduced MD5 hash passes 11 of the 15 tests using the linear sequence $1, 2, \ldots, n$ as input. For the full MD5 hash, Tzeng and Wei (2008) claim 15 out of 15 tests passed, though we are only able to reproduce 12 with certainty. In comparison, the standard C rand() function passes only six. Tzeng and Wei (2008) have a much more thorough analysis, and we refer the reader to their work for more information.

Our initial implementation generated a purely random sample over the entire surface of the light for each ray, but this resulted in somewhat grainy images, with the randomness being visible in the output even after applying the Gaussian filter. Consequently, we have moved to a jittered grid pattern when using tracing multiple rays; e.g., for four rays, the endpoint of each ray is chosen randomly from within one cell of a 2x2 grid overlaid onto the light. Figure 3.3 shows the difference between the two methods. Note the slightly noisier quality of the shadows in the image produced via pure random sampling.



(a) pure random sampling

(b) jittered grid sampling

FIG. 3.3. Effects of different sampling patterns. Images were rendered using two depth layers and hierarchical search, with nine rays per fragment.

We have found that for scenes with small lights and/or lighting angles that produce narrow penumbrae, a 2x2 grid produces visually acceptable results, while a 3x3 grid is sufficient in most other scenarios. Furthermore, scenes with highly detailed textures hide the majority of sampling artifacts, meaning a smaller grid may be used in such cases as well.

3.2 The Inadequacy of a Single Depth Map

The use of a single depth map causes several problems, as a single map does not provide enough information to adequately reconstruct the scene geometry. The most readily apparent effect of this is that only outer penumbrae are possible. If a fragment would be shadowed using conventional shadow mapping, it will always be shadowed by our method using a single depth map; that is, any ray cast to the fragment must pass behind a micro-occluder at its endpoint. Furthermore, an outer penumbra created this way often does not appear as a plausible shadow, because the probability of a point being shadowed is discontinuous, being 1.0 everywhere in the interior and less than 0.5 in the penumbra. The filtering stage is only able to hide the discontinuity for very small penumbrae.

Another problem is that severe aliasing occurs if the ray tracing is performed via linear search. While this may seem obvious, most applications of ray-height field intersection are able to refine linear samples once a boundary crossing is detected, using e.g. binary search. In single-layer shadowing, no such boundary crossing exists (or rather, it is the end result that we are searching for), so no refinement is possible. While hierarchical search methods solve the aliasing problem, linear search becomes viable if a second depth layer is introduced, and may be desirable due to its ease of implementation.

3.3 Ray Tracing with Two Depth Layers

Adding a second depth map can elegantly solve the problems described above, and we consider it the minimum for achieving quality results. Since the primary drawback of a single depth map is that it does not approximate occluder geometry well enough, the second map should be rendered from a point of view that provides the most additional information about the scene. A logical choice, then, would be to render depths from a point of view directly opposite the light, facing in the reverse direction. This would capture the surfaces farthest from the light, and would allow us to measure the sizes of objects in the scene in addition to their positions. Hence we refer to the result as the *far* depth layer, and the original one as *near*. Figure 3.4 illustrates how occluding geometry can be reconstructed using multiple depth layers. Both the far and near depth layers effectively form hulls around occluders, with the intersection of the two hulls being the reconstructed geometry. Stated another way, the intersection of the hulls is the closest approximation to the actual geometry that can be made using only the information contained in the depth maps. Note that information about the geometry may still be lost in the two-layer reconstruction, particularly for scenes with high depth complexity, in which gaps may exist between multiple occluders, for example. We return to this problem in Section 3.4.

Since placing the viewpoint opposite the light would require prior calculation of the extent of the scene, in our implementation we obtain a similar result by rendering the second depth map from the same point of view as the first, but with front face culling enabled and the z-buffer depth test reversed, thus capturing the back surfaces of the deepest objects in the scene. Since we are only interested in occluders, our method requires that “pure receivers” (such as the ground plane in our sample images) be removed from the scene when rendering depths, so as not to interfere with obtaining information about occluder geometry. Note that although we require classification of non-occluders, the remaining objects may

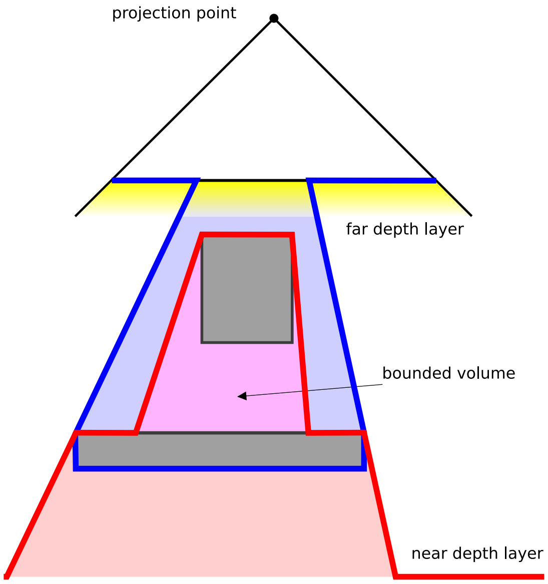


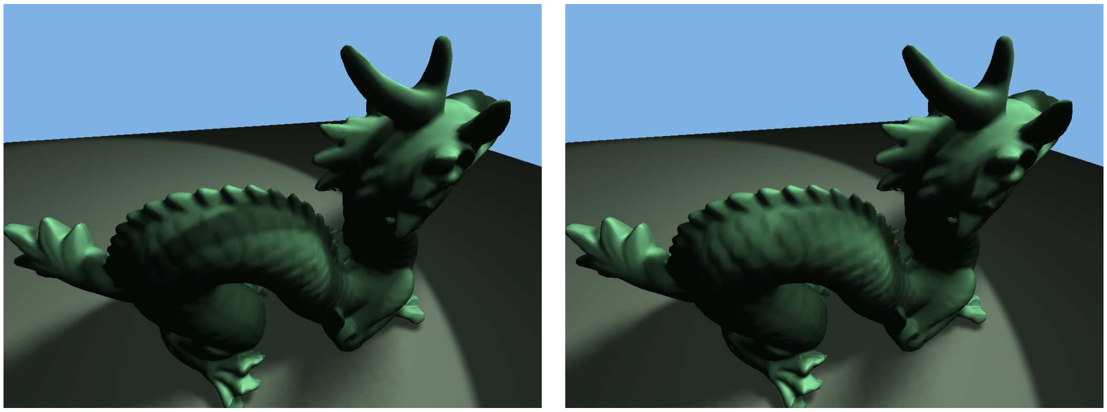
FIG. 3.4. Geometry reconstruction using two depth layers. Possible occluders are bounded between the near layer (red) and the far layer (blue). It is possible to use a third depth layer (not shown) to verify the existence of occluders inside the bounded volume.

still act as receivers, meaning that unlike other methods that use geometry classification, we retain support for self-shadowing.

We treat the two depth layers consistent with the assumption that they represent the boundaries of solid objects, so that when tracing rays, an intersection occurs when a ray passes anywhere between the two layers. If linear search is used, a useful idea is that any point along the ray, not between the two layers, may be classed as either in front of or behind any potential occluders. Transitions between the two states can then be treated as candidates for refinement using binary search, eliminating much (though not all) of the aliasing that exists when using a single layer. There is one caveat, however; because we generate the layers by first clearing the depths in the near layer to 1 and those in the far layer to 0, there can exist situations when a point is both in front of the near layer and behind the far layer. We class these points separately, and refine the search on any transitions *to* an “in front of” state or *from* a “behind” state. In cases when the the binary refinement does not detect an intersection, the algorithm returns to linear search at the farther point.

3.4 Additional Depth Layers

We treat the entire volume between the two depth layers as being occupied by a solid object. In reality, objects in the scene are likely to have hollow regions and concavities that cannot be represented by the depth layers. The visual effect of these details on the primary receiving surfaces in the scene is minor in most cases, but they can have a noticeable impact when occluders exhibit significant self-shadowing, as shown in Figure 3.5. With two depth layers, rays cast to points inside the bounded volume will always result in shadows, limiting self-shadowing to outer penumbrae as was described previously for single depth layers. The shadow cast by the dragon’s horn in Figure 3.5(a) is a result of points being inside the volume bounded by the two depth layers.



(a) two depth layers (b) three depth layers

FIG. 3.5. Self Shadowing. A third depth layer (b) allows for correct softening of the shadow cast by the dragon’s horn. Images were generated using 4 rays per fragment; (a) uses hierarchical search while (b) uses linear/binary.

An option for gaining more information about occluder geometry is to use depth peeling to generate additional depth layers parallel to the existing two. However, on current graphics hardware this would require another rendering pass for each layer, so we do not consider it practical. Instead, we note that we can make use of depths rendered from the point of view of the eye, since they must be generated eventually, regardless of whether or not we use them for shadowing. To do so, we reformulate our algorithm as an application of deferred shading, which executes according to Algorithm 1.

```
• Render the unshadowed scene from the eye point, storing colors
  and depths in separate buffers.
• Render the near and far depth layers from the point of view of
  the light.
• Run a shader over the eye-space depth buffer:
  – Recover the world space coordinates of each fragment based on
    the depth and texture coordinates.
  – Trace shadow rays using all three available depth maps.
  – Write shadow values to an output buffer.
• Filter the shadow buffer.
• Combine the shadow buffer with the color buffer.
```

Algorithm 1: Our shadowing technique formulated using deferred shading and three depth maps.

With the linear search model that we have described up to this point, it is a simple matter to make use of the eye-space depth map; any time a sample point along the ray would be inside the volume bounded by the near and far depth layers, instead of declaring an intersection, test if the point is visible from the eye and declare an intersection only if it is not. In Figure 3.5(b), this process results in a softer shadow cast by the dragon's horn.

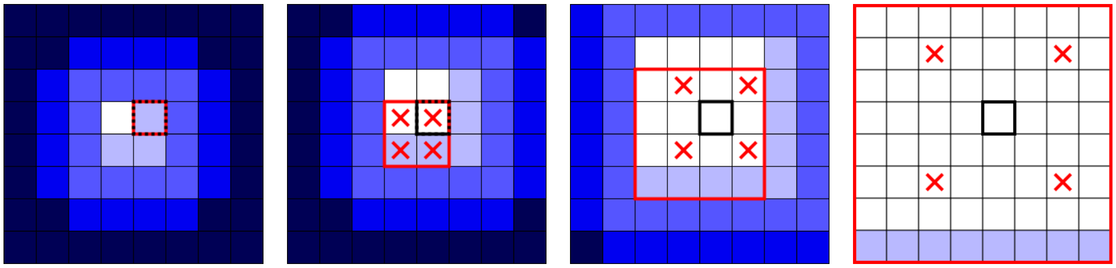
Coincidentally, we observe a significant performance increase when using deferred shading, since no ray tracing is wasted on fragments that will later be z-culled.

3.5 Hierarchical Search

Until this point we have focused our discussion on linear/binary search as the method for tracing rays, since it is the simplest approach to implement. However, it does come with significant drawbacks. The linear stage suffers from aliasing and the performance is highly inconsistent, dropping off severely when the binary refinement stage does not find an intersection on the first try. As an alternative, we have implemented ray tracing using a hierarchical data structure that replaces the near and far depth layers. This both eliminates aliasing and provides better performance.

Our data structure is most directly based on the N-buffer (Décoret 2005), although we make some modifications to it in the same manner as Schwarz and Stamminger (2007). To briefly explain, the N-buffer is a texture stack containing $log_2(w) + 1$ levels for square textures of width w. A texel at level n stores the maximum value of the surrounding texels in a neighborhood of size 2^n at level 0. We use the terms *fine* and *coarse* to refer to levels with smaller and larger neighborhoods, since the structure bears some resemblance to a mipmap. In Décoret's (2005) original formulation, texels were located at the lower-left corner of their neighborhoods; we move them approximately to the center, which allows the buffer to provide more information near texture edges. Instead of storing a simple maximum, we use multi-channel textures to store the minimum depth of the near depth layer (z_{near}^{min}) and the maximum depth of the far layer (z_{far}^{max}). We also store the maximum depth of the near layer (z_{near}^{max}), though this is not directly used in ray tracing and will be explained in the next section. Each level in the N-buffer can be efficiently constructed by running a shader that finds the min or max of four samples from the previous level, with level 0 being copies of the original depth maps. Figure 3.6 illustrates this process.

We use Algorithm 2 to trace rays through the N-buffer. Since we trace rays from the light to the fragment, the primary interaction is with z_{near}^{min}. We use the term *stepping* to



(a) Level 0 (b) Level 1 (c) Level 2 (d) Level 3

FIG. 3.6. A four-level N-buffer. A texel at level n stores the maximum luminosity of the surrounding texels in a neighborhood of size 2^n at level 0. The texels highlighted in black have neighborhoods highlighted in red, and were calculated by taking the maximum of the four texels from the previous level, at the locations marked with red Xs.

refer to the portion of the ray that has been searched; e.g., the ray start point is at stepping 0 and the end point is at stepping 1. For readability, terminal cases are described separately.

Ascending in the N-buffer is optional, but we have found it useful, since otherwise the algorithm can in rare cases spend many iterations stepping one texel at a time at the finest level. Terminal cases are as follows:

- If the ray passes between z_{near}^{min} and z_{far}^{max} at level 0 of the N-buffer, an intersection has occurred.
- If at any point the stepping advances beyond the endpoint of the ray, no intersection can occur.

A few difficulties arise in the implementation of this algorithm, which may not be apparent at first. The ray stepping is not aligned to texel boundaries in the N-buffer, so when calculating neighborhood sizes the actual width used is $2^n - 1$, plus some fractional value to align to the nearest texel boundary. Also, because our depth maps are created using perspective projections, the stepping has different values in world space and texture space, so frequent conversion is required. These two facts, along with the overall structure of the algorithm, lead to much more complicated shader code relative to linear/binary search. Even so, the faster convergence of hierarchical search yields noticeably better performance.

3.6 Penumbra Classification

Since we typically cast several rays for each shaded fragment, there can be significant overhead even if every ray is traced quickly. It is therefore advantageous to classify fragments as fully shadowed, fully lit, or potentially in penumbra, since in the former two cases we can avoid casting any rays at all. We employ a method similar to Guennebaud et al. (2006), in which we find the minimum and maximum occluder depth in the near depth layer (z_{near}^{min} and z_{near}^{max}) that can be encountered by any possible ray. We then classify according to the fragment depth, z:

- If $z \leq z_{near}^{min}$, then nothing can occlude the fragment and it is fully lit.
- If $z \geq z_{near}^{max}$, then every possible ray will hit an occluder, so the fragment is fully shadowed.

Note that the second condition becomes invalid when using three depth maps, since rays are allowed to travel *between* occluders. In this case we only test the first condition.

Values for z_{near}^{min} and z_{near}^{max} can easily be obtained from the coarsest level of the N-buffer; however, these values cover the entire scene and tend to not lead to useful classification. Instead, we use the value of z_{near}^{min} as the initial step in an iterative refinement scheme (again, similarly to Guennebaud et al. (2006)). The region of the depth map that might be sampled by any possible rays can be calculated using similar triangles, as shown in Figure 3.7, or according to Equation 3.3; n is the distance from the light to the projection point and $size$ is a fraction of the depth map width.

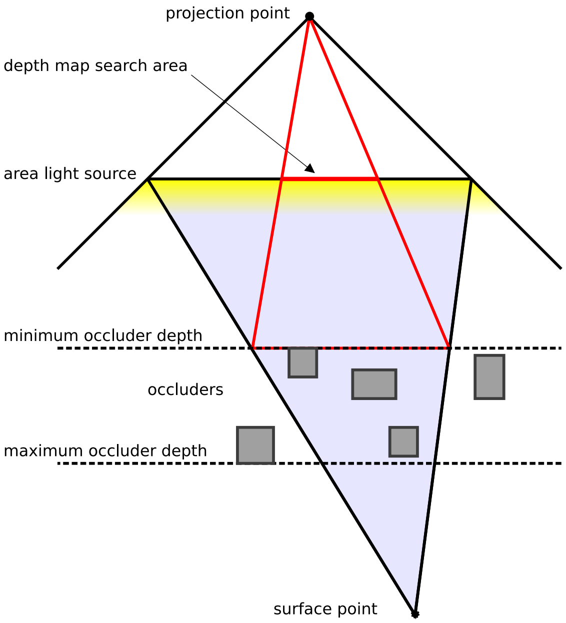


FIG. 3.7. Determination of the portion of the depth map containing potential occluders. All possible rays from a point in the scene intersect a plane at the minimum occluder depth. The intersection region projects onto a subset of the depth map.

$$\mathit{size} = \frac{n(z - z_{near}^{min})}{z_{near}^{min}(z - n)} \tag{3.3}$$

We use the computed region to find a less conservative value of z_{near}^{min} by sampling a finer level of the N-buffer. Two iterations are generally sufficient to produce a tight bound around penumbrae in the scene. Figure 3.8 shows the results of this method.

3.7 Other Acceleration Techniques

When casting rays, it is possible in many cases to immediately classify a ray as intersecting or non-intersecting and thus avoid having to perform expensive calculations for it. Figure 3.8 shows these cases in green, and we identify three scenarios in which they occur:

- If the surface normal faces opposite the ray (i.e. their dot product is negative), the ray

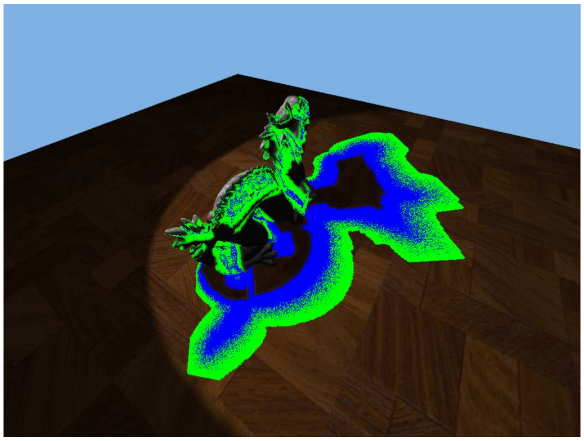


FIG. 3.8. Classification of fragments. Green areas are potentially in penumbra, while full calculations are only performed in blue areas.

is considered intersecting and results in a shadow. With small area lights, performing this test on a per fragment basis rather than per ray may produce acceptable results, since all possible rays would have similar directions.

- If the endpoint of a ray is inside a solid object, the ray must intersect an occluder. While rays are never *actually* created with endpoints inside objects, this test is useful when using two depth layers, as it catches instances of apparent self-shadowing early, using only a single texture fetch at the finest level of the N-buffer. Due to biasing, this test also applies to back-facing surfaces and is useful if the first test cannot be applied (e.g. with deferred shading, we might want to avoid rendering normals).
- Lastly, we perform the fragment classification discussed previously, but using a region in the depth map that contains the projection of the current ray, rather than that of all possible rays. This region is potentially much smaller, yielding a more accurate determination of whether or not the ray can intersect an occluder.

The last scenario can be made more useful by shortening the rays, thus making their projections smaller. We shorten the rays by advancing their endpoints up to z_{far}^{max}, and their start points down to z_{near}^{min}, as determined in the fragment classification step. This process is shown in Figure 3.9. For rays that require full intersection calculations, ray shortening produces faster convergence to a solution, since the starting point in the N-buffer is moved towards the finer levels.

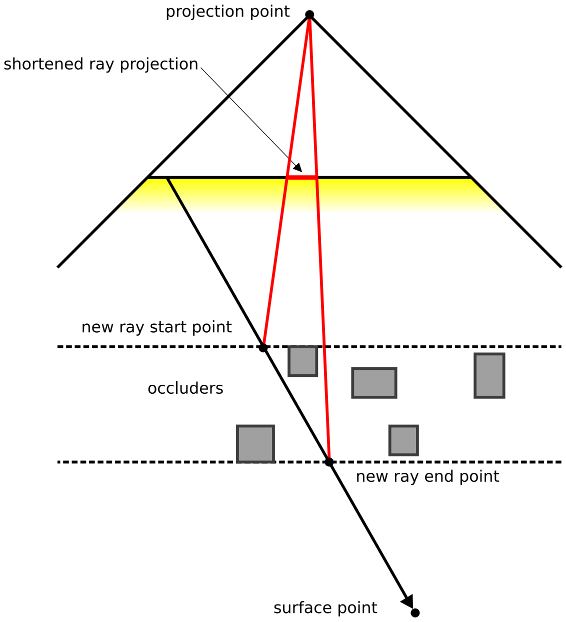


FIG. 3.9. Ray shortening. Ray endpoints are modified based on the bounds for possible occluder depths, resulting in a smaller affected region of the depth map.

```
• Determine an initial level in the N-buffer such that a neighborhood is able to contain the entire texture-space projection of the ray.
• Find $z_{near}^{min}$ for the neighborhood containing the ray.
• Set the initial stepping to the point where the ray intersects the plane at $z_{near}^{min}$.
• Descend one level in the N-buffer and place the neighborhood in texture space such that it contains the intersection point and as much of the *forward* portion of the ray as possible.
• Repeat the following until terminal cases are reached:
  – Find $z_{near}^{min}$ and $z_{far}^{max}$ in the current neighborhood.
  – If, in the current neighborhood, a portion of the ray passes between $z_{near}^{min}$ and $z_{far}^{max}$:
    ∗ Find the point where the ray intersects the plane at $z_{near}^{min}$.
    ∗ If the intersection point is farther along the ray than the current stepping, advance the stepping to the intersection point.
    ∗ Descend one level in the N-buffer, and adjust the current neighborhood to contain the forward portion of the ray.
  – Otherwise, the current portion of the ray does not intersect anything, so:
    ∗ Advance the ray stepping a distance equal to the current neighborhood size.
    ∗ Ascend one level in the N-buffer, and adjust the current neighborhood to contain the forward portion of the ray.
```

Algorithm 2: Hierarchical search using an N-buffer

Chapter 4

RESULTS

We have implemented two versions of our algorithm, one using linear/binary search and three depth layers, and one using hierarchical search with two layers. By default, both use deferred shading, although we also present performance numbers for the hierarchical search version with deferred shading disabled. Also, while the linear/binary search implementation does not use the N-buffer directly for ray tracing, it does make use of all the associated acceleration techniques. We consider the N-buffer (or an equivalent data structure) essential to our technique. Figure 1.1 shows some sample renderings using our method, and Figures 4.1 and 4.2 show some additional cases with texturing disabled to better show any artifacts. The reference images in both cases were generated by averaging 4096 point lights (about 40 seconds per frame). Note the broad shadows cast against against the far wall by the dragon in Figure 1.1(a), or cast on the floor by the Buddha statue in Figure 1.1(b). Both cases correctly show relatively sharp contact shadows where the objects touch the floor. Though some minor discrepancies are highlighted in Figures 4.1 and 4.2, our results (particularly Figures 4.1(b) and 4.2(b)) are largely indistinguishable from the reference images, especially when compared against the results for conventional shadow mapping. Also note the darkening of the shadow directly beneath the elongated cube in Figures 4.2(a) and 4.2(b); this is a difficult case for other shadowing algorithms, since the shadow is a result of the object's extent perpendicular to the light, which cannot be determined with a single shadow map.

4.1 Limitations

Being based on ray tracing, our method has the potential to give physically accurate shadows. However, there are a few cases where discrepancies can appear. We require that the depth maps be able to adequately represent occluder geometry, and while this is most often the case, artifacts can arise in the form of over-dark self shadowing for complex occluders. Figure 3.5 specifically shows this effect, although the hierarchical results in Figures 4.1(a) and 4.1(b) also contain a moderate example of it. Compare the blue highlighted areas to those in Figures 4.1(c) and 4.1(d); in the former, the umbrae are slightly overextended.

Other artifacts are related to our post-filtering process. In scenes with especially wide penumbrae, our method can produce uneven looking shadows due to undersampling of the light source visibility, in which case the filtering is insufficient. The highlighted region of Figure 4.2(a) demonstrates this effect. The opposite case can occur as well, in which the filtering process excessively blurs fine details in the shadow. The shadow of the dragon's horn in Figures 4.1(a), 4.1(b), and 4.1(c) (highlighted in red) is a good example; the shadow in the reference image (Figure 4.1(d)) is thinner and darker in comparison.

All of these problems can be dealt with by simply adding more depth layers or casting more rays (which would reduce the need for post-filtering). Both come at a cost in performance, and while we have found that casting four or nine rays per fragment represents a good balance between visual quality and computational expense, it is conceivable that with future hardware this balance might shift upwards.

4.2 Performance

Performance results were obtained on an Nvidia 8800 GTS (320 MB) with driver version 180.11, running on Ubuntu Linux 8.10. Our implementation uses OpenGL 2.1 with shaders written in GLSL. We tested two scenes, the dragon shown in Figure 4.1 and the elongated cube shown in 4.2. The dragon model contains 100,000 triangles, while the cube shows the effects of reduced depth complexity and also eliminates any bottlenecks due to triangle count.

Tables 4.1 and 4.2 show frame rates at various shadow map resolutions, for the two scenes we tested. Figure 4.3 summarizes the results graphically for the hierarchical search implementation with deferred shading, which is the best version by a large margin. The hierarchical / deferred shading frame rates are well within the range that should be considered real-time, particularly with a shadow map resolution of 512x512 or less. We consider 512x512 to be the ideal resolution for our algorithm on current hardware; performance does not improve dramatically at lower resolutions when rendering relatively complex scenes such as the dragon, while at much lower resolutions temporal aliasing becomes a problem. Note that the shadow map resolution is limited to 1024x1024 and below due to memory constraints on the N-buffer.

Tables 4.3 and 4.4 show results at various screen resolutions, with Figure 4.4 summarizing the hierarchical / deferred shading results graphically. The performance drop-off with increasing resolution should be expected, since the screen resolution directly affects the number of rays that must be traced. However, frame rates remain usable at all resolutions tested, up to 1280x1024.

Of the two search methods, linear/binary search is more sensitive to geometric complexity, due to the fact that as more detail appears in the depth layers, the algorithm attempts binary refinement more often. Hierarchical search is also somewhat sensitive to geomet-

Search Type	Rays	Deferred Shading	Shadow Map Resolution		
			256x256	512x512	1024x1024
hierarchical	2x2	yes	46	40	28
	3x3		37	34	21
	2x2	no	30	26	18
	3x3		18	16	12
linear/binary	2x2	yes	25	18	15
	3x3		12	11	10

Table 4.1. Performance scaling with shadow map resolution for the dragon scene (Figure 4.1). Screen resolution is 1024x768. Values are in frames per second.

Search Type	Rays	Deferred Shading	Shadow Map Resolution		
			256x256	512x512	1024x1024
hierarchical	2x2	yes	70	50	35
	3x3		60	45	31
	2x2	no	66	47	34
	3x3		50	44	30
linear/binary	2x2	yes	36	32	23
	3x3		25	22	17

Table 4.2. Performance scaling with shadow map resolution for the elongated cube scene (Figure 4.2). Screen resolution is 1024x768. Values are in frames per second.

ric complexity, since increased detail causes it to spend more iterations stepping through the finer levels of the N-buffer. Although not reflected in the numbers, performance with linear/binary search is very inconsistent; we believe hierarchical search should be the preferred implementation, even ignoring average frame rates. As should be expected, deferred shading is more beneficial in complex scenes, nearly doubling the frame rate in a few cases. The benefits of deferred shading vastly outweigh the cost of an additional rendering pass in screen-space, so it should be preferred in any implementation as well.

The results show that hierarchical search is more sensitive to depth map resolution,

Search Type	**Rays**	**Deferred Shading**	**Screen Resolution**		
			800x600	1024x768	1280x1024
hierarchical	2x2	yes	45	40	32
	3x3		40	34	22
	2x2	no	33	26	19
	3x3		20	16	12
linear/binary	2x2	yes	27	18	14
	3x3		17	11	8

Table 4.3. Performance scaling with screen resolution for the dragon scene (Figure 4.1). Shadow map resolution is 512x512. Values are in frames per second.

Search Type	**Rays**	**Deferred Shading**	**Screen Resolution**		
			800x600	1024x768	1280x1024
hierarchical	2x2	yes	74	50	41
	3x3		61	45	34
	2x2	no	70	47	41
	3x3		49	44	35
linear/binary	2x2	yes	41	32	23
	3x3		31	22	15

Table 4.4. Performance scaling with screen resolution for the cube scene (Figure 4.2). Shadow map resolution is 512x512. Values are in frames per second.

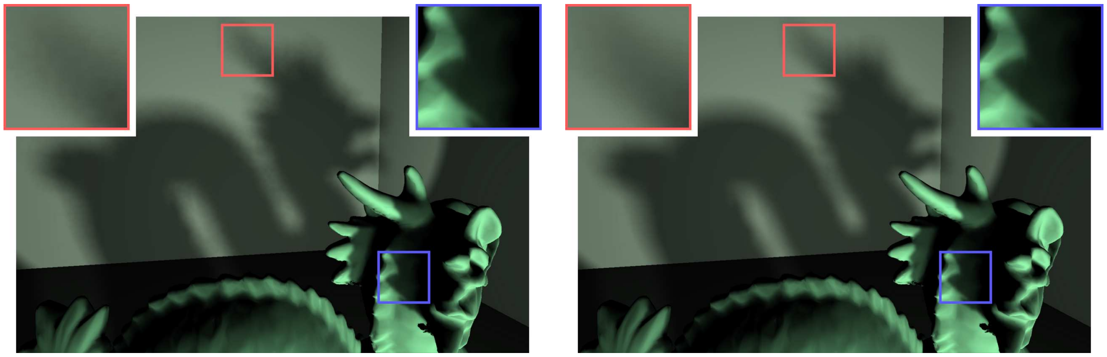
but a reason for this is that we used a fixed step size for the linear portion of linear/binary search; that is, hierarchical search becomes more accurate at higher depth map resolutions while our implemention of linear/binary search does not. Both methods are affected by time required to construct the N-buffer, which is likely the largest component of depth map resolution sensitivity. Caching is also a factor; while our Monte Carlo sampling causes completely random texture access, the ray shortening procedure described previously should keep the accesses within a small neighborhood of each other, thus preventing at least some cache misses. This effect is reduced with increased texture size.

Generally speaking, our performance numbers are on par with current methods for rendering physically plausible soft shadows, e.g. backprojection (Guennebaud, Barthe, & Paulin 2006) or bitmask (Schwarz & Stamminger 2007) soft shadowing (the former being slightly faster than our method, and the latter slightly slower). This is not surprising, since we use many of the same acceleration techniques. However, both of the aforementioned methods treat occluders as sets of planar patches or quads, rather than solid volumes as in our approach; hence neither method is able to make good use of the additional information provided by multiple depth maps. Furthermore, both must take special measures to prevent problems with overlaps or gaps between micro-occluders; in the case of backprojection, this leads to unavoidable over-darkening. Ray tracing naturally avoids such problems.

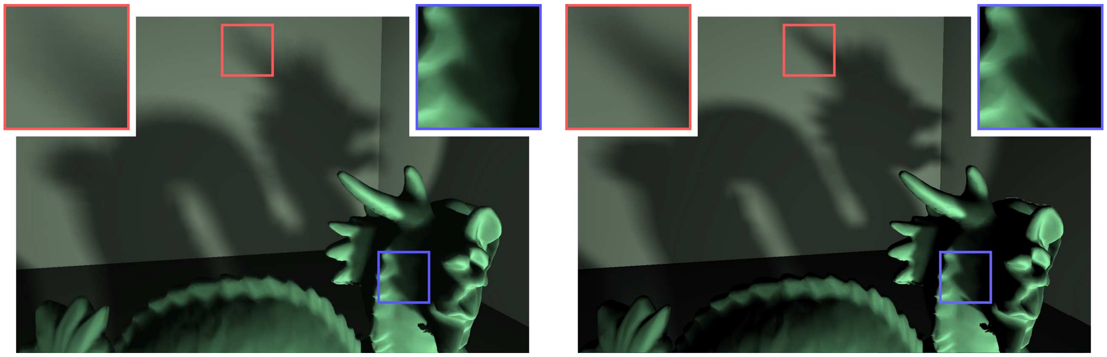
4.3 Impact of the MD5 Hash

Our reliance on the reduced MD5 hash for random number generation introduces an unknown factor into our algorithm's performance, since it differs from the usual practice of using a handful of carefully selected fixed sample positions to represent a jittered grid. However, in our testing the hash did not incur a significant performance penalty. Removing it from the code entirely resulted in a frame rate increase of 2-3 frames per second at most (less than 10%), and 0 in many cases. While the hash does involve a relatively large amount of computation, even for the reduced 16-round version, our implementation does not involve any looping or branching, and requires few registers. Consequently it maps well to the capabilities of GPU hardware.

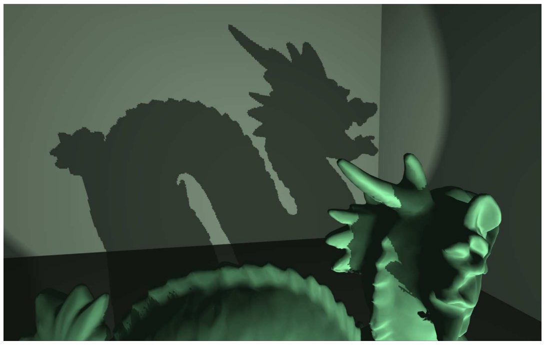
In comparison, it is very unlikely that using pre-generated sample positions would offer better quality, since even if the samples were created by a high-quality random number generator, it is unlikely to be noticeably better than MD5 (we refer to the DIEHARD results described in Section 3.1). Furthermore, a very large number of distinct sample points would be required. It is also worth noting that using fixed sample points does not map as well to GPU hardware, since the process must involve a memory lookup. Current hardware favors pure computation over memory access, and the gap between the two is likely to widen in the future.



(a) hierarchical, 2x2 (b) hierarchical, 3x3

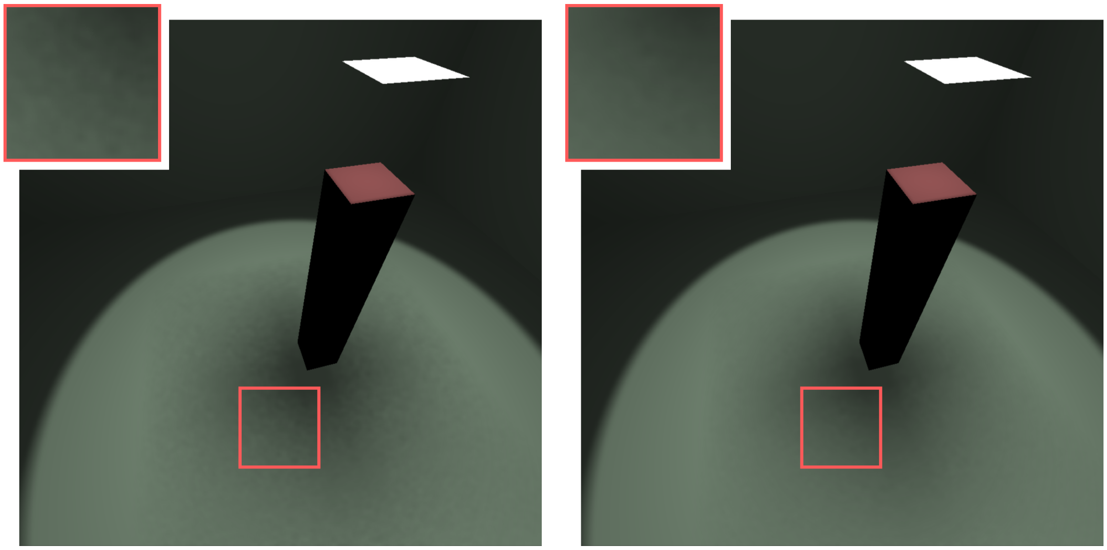


(c) linear/binary, 3x3 (d) reference image

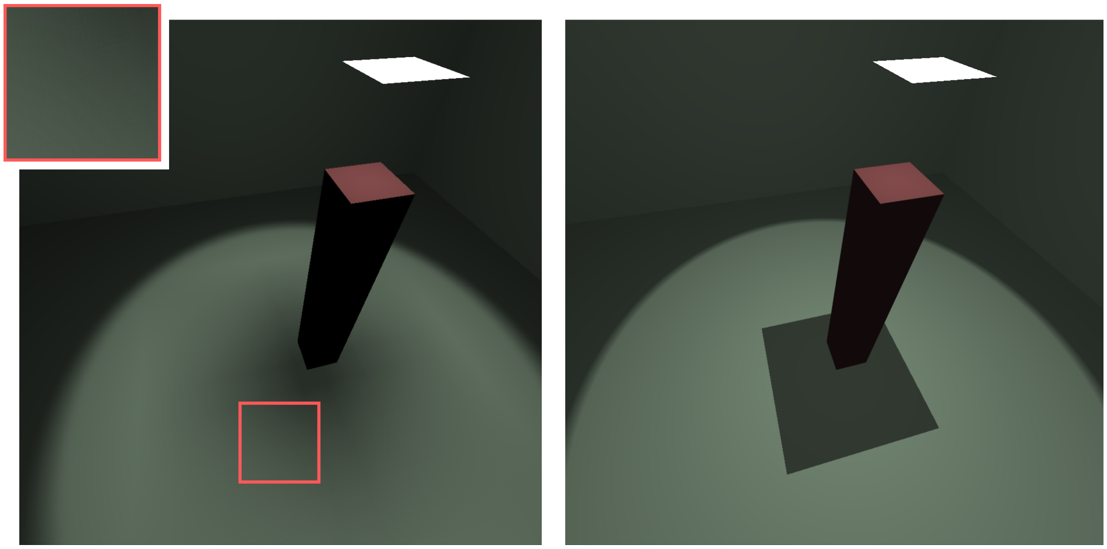


(e) conventional shadow mapping

FIG. 4.1. Shadowing of the dragon model. Labels show the search type and number of rays per fragment. Shadow map size is 512x512.



(a) hierarchical, 2x2 (b) hierarchical, 3x3



(c) reference image (d) conventional shadow mapping

FIG. 4.2. Shadowing of an elongated cube, a difficult case for many soft shadowing algorithms. Labels show the search type and number of rays per fragment. Shadow map size is 512x512.

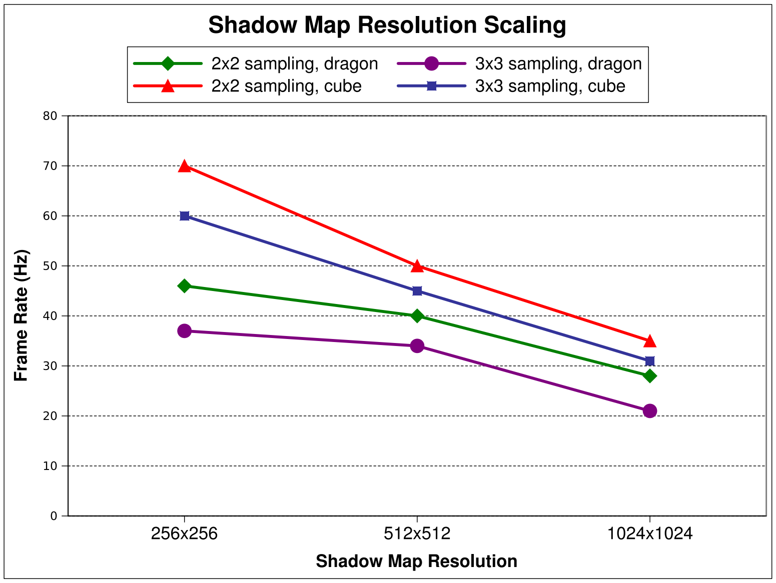


FIG. 4.3. Shadow map resolution scaling for hierarchical search with deferred shading enabled. Labels show the number of rays per fragment and which scene was used.

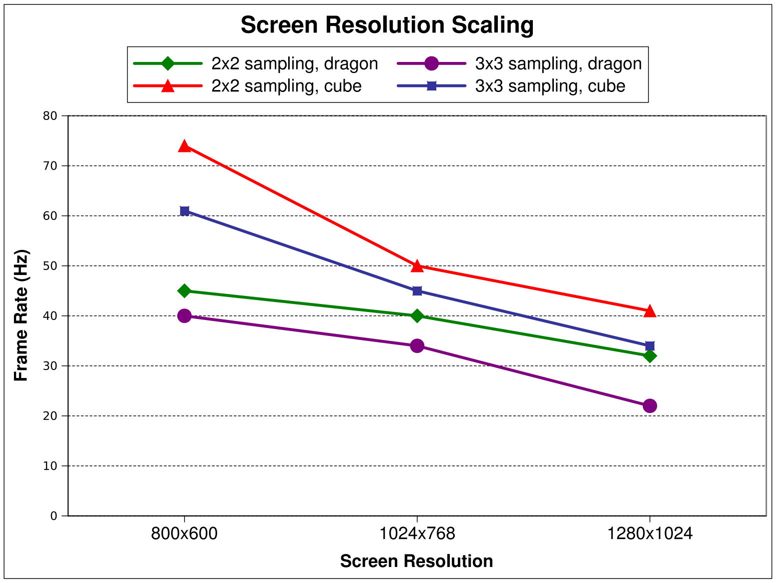


FIG. 4.4. Screen resolution scaling for hierarchical search with deferred shading enabled. Labels show the number of rays per fragment and which scene was used.

Chapter 5

CONCLUSION

We have demonstrated physically accurate soft shadowing in real-time, via stochastic ray tracing of depth maps. We believe that ray tracing in this manner offers a higher visual quality than current state of the art real-time shadowing algorithms, and we have found that it can be accomplished with similar levels of performance on current graphics hardware. Furthermore, since our work is based on conventional shadow mapping, it should be possible to implement it within existing software frameworks.

Our technique relies on depth maps to approximate occluder geometry, and we consider two depth maps the minimum to achieve a good reconstruction, which is necessary for high quality shadows. Additional depth maps are beneficial, but returns diminish rapidly beyond the second or third. We also consider the use of a hierarchical data structure essential for accelerating the algorithm; while such a structure is directly useful for tracing individual rays, we also use it to eliminate a large number of rays entirely.

5.1 Future Work

Textured light sources are possible with our method. Preliminary work indicates that visually accurate results can be obtained by allowing each ray to sample a low frequency texture on the surface of the light and return a luminosity value, rather than a simple binary shadowed or lit determination. Going farther, it should be possible for each ray to return an RGB value corresponding to the received luminosity on each color channel. This would make our technique useful for environment mapping, for example.

There also exists the possibility of making alterations to the N-buffer; one direction could be to make use of the fact that the neighborhood sizes at each level need not be powers of two. Some investigation would be needed to determine if an optimal formula exists for the neighborhoods, but at the very least, it should be possible to decrease the number of necessary levels by using e.g. powers of three. This would reduce the memory requirement, which is the N-buffer's main drawback. We allow that for practical purposes, others may want to implement our technique using more conventional image pyramids, and it remains to be seen how much of an impact on performance this would have.

We also expect that other performance improvements to the algorithm are possible. Of note is the fact that our depth maps were all created using perspective projection, while orthogonal projection should in theory work just as well for the purpose of ray tracing. This would simplify the shader code a great deal by eliminating the need to convert the ray stepping between texture and world space. However, a more interesting possibility is that multiple lights may be able to share a single orthogonally projected shadow map; one could imagine a framework in which a few axis-aligned depth maps are generated for an arbitrary number of lights throughout the scene.

Appendix A

REDUCED MD5 HASH IMPLEMENTATION

The following is our implementation of the reduced 16-round MD5 hash in GLSL. Screen coordinates are used for the seed value. Following Tzeng and Wei’s (2008) example, we allow the use of a key value to further scramble the seed.

```
uvec4 md5rand(uvec4 seed, unsigned int key)
{
    unsigned int d00 = seed.x ^ key;
    unsigned int d01 = seed.y ^ key;
    unsigned int d02 = seed.z ^ key;
    unsigned int d03 = seed.w ^ key;

    // Values commented out as they do not
    // actually affect the calculation

    //unsigned int d04 = 0x80000000u;
    //unsigned int d05 = 0u;
    //unsigned int d06 = 0u;
    //unsigned int d07 = 0u;
    //unsigned int d08 = 0u;
    //unsigned int d09 = 0u;
    //unsigned int d10 = 0u;
    //unsigned int d11 = 0u;
    //unsigned int d12 = 0u;
    //unsigned int d13 = 0u;
    //unsigned int d14 = 0u;
    //unsigned int d15 = 0x80u;
```

```
uvec4 digest = uvec4(0x01234567u, 0x89ABCDEFu,
                     0xFEDCBA98u, 0x76543210u);
uvec4 tD = digest;
unsigned int Ft;
unsigned int rot_temp;

Ft = (tD.x & tD.y) | ((˜tD.x) & tD.z);

// The number on the next line is sin(1) * 2ˆ32.
// Likewise, the constants in the other blocks are
// sin(2), sin(3), etc.

rot_temp = tD.x + Ft + d00 + 0xD76AA478u;
tD.x = tD.y + (rot_temp << 7u) + (rot_temp >> 25u);
tD = tD.yzwx;
digest += tD;

Ft = (tD.x & tD.y) | ((˜tD.x) & tD.z);
rot_temp = tD.x + Ft + d01 + 0xE8C7B757u;
tD.x = tD.y + (rot_temp << 12u) + (rot_temp >> 20u);
tD = tD.yzwx;
digest += tD;

Ft = (tD.x & tD.y) | ((˜tD.x) & tD.z);
rot_temp = tD.x + Ft + d02 + 0x242070DBu;
tD.x = tD.y + (rot_temp << 17u) + (rot_temp >> 15u);
tD = tD.yzwx;
digest += tD;

Ft = (tD.x & tD.y) | ((˜tD.x) & tD.z);
rot_temp = tD.x + Ft + d03 + 0xC1BDCEEFu;
tD.x = tD.y + (rot_temp << 22u) + (rot_temp >> 10u);
tD = tD.yzwx;
digest += tD;

////////////////////////////////////////////////////

Ft = (tD.z & tD.x) | ((˜tD.z) & tD.y);
rot_temp = tD.x + Ft + d01 + 0xF61E2562u;
tD.x = tD.y + (rot_temp << 5u) + (rot_temp >> 27u);
tD = tD.yzwx;
digest += tD;
```

```
Ft = (tD.z & tD.x) | ((˜tD.z) & tD.y);
rot_temp = tD.x + Ft /* + d06 */ + 0xC040B341u;
tD.x = tD.y + (rot_temp << 9u) + (rot_temp >> 23u);
tD = tD.yzwx;
digest += tD;

Ft = (tD.z & tD.x) | ((˜tD.z) & tD.y);
rot_temp = tD.x + Ft /* + d11 */ + 0x265E5A52u;
tD.x = tD.y + (rot_temp << 14u) + (rot_temp >> 18u);
tD = tD.yzwx;
digest += tD;

Ft = (tD.z & tD.x) | ((˜tD.z) & tD.y);
rot_temp = tD.x + Ft + d00 + 0xE9B6C7ABu;
tD.x = tD.y + (rot_temp << 20u) + (rot_temp >> 12u);
tD = tD.yzwx;
digest += tD;

////////////////////////////////////////////////////

Ft = tD.x ˆ tD.y ˆ tD.z;
rot_temp = tD.x + Ft /* + d05 */ + 0xFFFA3942u;
tD.x = tD.y + (rot_temp << 4u) + (rot_temp >> 28u);
tD = tD.yzwx;
digest += tD;

Ft = tD.x ˆ tD.y ˆ tD.z;
rot_temp = tD.x + Ft /* + d08 */ + 0x8771F681u;
tD.x = tD.y + (rot_temp << 11u) + (rot_temp >> 21u);
tD = tD.yzwx;
digest += tD;

Ft = tD.x ˆ tD.y ˆ tD.z;
rot_temp = tD.x + Ft /* + d11 */ + 0x6D9D6122u;
tD.x = tD.y + (rot_temp << 16u) + (rot_temp >> 16u);
tD = tD.yzwx;
digest += tD;

Ft = tD.x ˆ tD.y ˆ tD.z;
rot_temp = tD.x + Ft /* + d14 */ + 0xFDE5380Cu;
tD.x = tD.y + (rot_temp << 23u) + (rot_temp >> 9u);
tD = tD.yzwx;
digest += tD;
```

```
    ////////////////////////////////////////////////////

    Ft = tD.y ˆ (tD.x | (˜tD.z));
    rot_temp = tD.x + Ft + d00 + 0xF4292244u;
    tD.x = tD.y + (rot_temp << 6u) + (rot_temp >> 26u);
    tD = tD.yzwx;
    digest += tD;

    Ft = tD.y ˆ (tD.x | (˜tD.z));
    rot_temp = tD.x + Ft /* + d07 */ + 0x432AFF98u;
    tD.x = tD.y + (rot_temp << 10u) + (rot_temp >> 22u);
    tD = tD.yzwx;
    digest += tD;

    Ft = tD.y ˆ (tD.x | (˜tD.z));
    rot_temp = tD.x + Ft /* + d14 */ + 0xAB9423A7u;
    tD.x = tD.y + (rot_temp << 15u) + (rot_temp >> 17u);
    tD = tD.yzwx;
    digest += tD;

    Ft = tD.y ˆ (tD.x | (˜tD.z));
    rot_temp = tD.x + Ft /* + d05 */ + 0xFC93A039u;
    tD.x = tD.y + (rot_temp << 21u) + (rot_temp >> 11u);
    tD = tD.yzwx;
    digest += tD;

    return digest;
}
```

REFERENCES

[1] Agrawala, M.; Ramamoorthi, R.; Heirich, A.; and Moll, L. 2000. Efficient image-based methods for rendering soft shadows. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 375–384. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

[2] Akenine-Möller, T., and Assarsson, U. 2002. Approximate soft shadows on arbitrary surfaces using penumbra wedges. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, 297–306. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association.

[3] Annen, T.; Mertens, T.; Bekaert, P.; Seidel, H.-P.; and Kautz, J. 2007. Convolution shadow maps. In Kautz, J., and Pattanaik, S., eds., *Rendering Techniques 2007: Eurographics Symposium on Rendering*, volume 18 of *Eurographics / ACM SIGGRAPH Symposium Proceedings*, 51–60. Grenoble, France: Eurographics.

[4] Annen, T.; Dong, Z.; Mertens, T.; Bekaert, P.; Seidel, H.-P.; and Kautz, J. 2008. Real-time, all-frequency shadows in dynamic scenes. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, 1–8. New York, NY, USA: ACM.

[5] Atty, L.; Holzschuch, N.; Lapierre, M.; Hasenfratz, J.-M.; Hansen, C.; and Sillion, F. 2006. Soft shadow maps: Efficient sampling of light source visibility. *Computer Graphics Forum* 25(4):725–741.

[6] Brabec, S., and peter Seidel, H. 2002. Single sample soft shadows using depth maps. In *In Graphics Interface*, 219–228.

[7] Chan, E., and Durand, F. 2003. Rendering fake soft shadows with smoothies. In *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering*, 208–218. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association.

[8] Chun, Y.; Oh, K.; and Kim, H. 2008. Multi-layer pyramidal displacement mapping. In *VRCAI '08: Proceedings of The 7th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, 1–2. New York, NY, USA: ACM.

[9] Cook, R. L.; Porter, T.; and Carpenter, L. 1984. Distributed ray tracing. *SIGGRAPH Comput. Graph.* 18(3):137–145.

[10] Crow, F. C. 1977. Shadow algorithms for computer graphics. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, 242–248. New York, NY, USA: ACM.

[11] D'Agostino, R. B., and Stephens, M. A., eds. 1986. *Goodness-of-fit techniques*. New York, NY, USA: Marcel Dekker, Inc.

[12] Décoret, X. 2005. N-buffers for efficient depth map query. *Computer Graphics Forum* 24(3):393–400.

[13] Donnelly, W., and Lauritzen, A. 2006. Variance shadow maps. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, 161–165. New York, NY, USA: ACM.

[14] Dummer, J. 2006. Cone step mapping: An iterative ray-heightfield intersection algorithm. Available online at http://www.lonesock.net/files/ConeStepMapping.pdf [Accessed: 1 May 2009].

[15] Fernando, R. 2005. Percentage-closer soft shadows. In *SIGGRAPH ’05: ACM SIGGRAPH 2005 Sketches*, 35. New York, NY, USA: ACM.

[16] Forest, V.; Barthe, L.; and Paulin, M. 2006. Realistic soft shadows by penumbra-wedges blending. In *GH ’06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 39–46. New York, NY, USA: ACM.

[17] Guennebaud, G.; Barthe, L.; and Paulin, M. 2006. Real-time soft shadow mapping by backprojection. In *Eurographics Symposium on Rendering (EGSR), Nicosia, Cyprus, 26/06/2006-28/06/2006*, 227–234. http://www.eg.org/: Eurographics.

[18] Hasenfratz, J.-M.; Lapierre, M.; Holzschuch, N.; and Sillion, F. 2003. A survey of real-time soft shadows algorithms. *Computer Graphics Forum* 22(4):753–774.

[19] Heidrich, W.; Brabec, S.; and Seidel, H.-P. 2000. Soft shadow maps for linear lights. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, 269–280. London, UK: Springer-Verlag.

[20] Kajiya, J. T. 1986. The rendering equation. In *SIGGRAPH ’86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 143–150. New York, NY, USA: ACM.

[21] Lauritzen, A. 2007. Summed-area variance shadow maps. In Nguyen, H., ed., *GPU Gems 3*. Addison-Wesley Professional. 157–182.

[22] Marsaglia, G. 1995. The Marsaglia random number CDROM including the diehard battery of tests of randomness. Available online at http://www.stat.fsu.edu/pub/diehard/ [Accessed: 1 May 2009].

[23] NVIDIA. 2006. Nvidia geforce 8800 GPU architecture overview. Technical Brief.

[24] Oh, K.; Ki, H.; and Lee, C.-H. 2006. Pyramidal displacement mapping: a GPU based artifacts-free ray tracing through an image pyramid. In *VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology*, 75–82. New York, NY, USA: ACM.

[25] Policarpo, F., and Oliveira, M. M. 2006. Relief mapping of non-height-field surface details. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, 55–62. New York, NY, USA: ACM.

[26] Policarpo, F., and Oliveira, M. M. 2007. Relaxed cone stepping for relief mapping. In Nguyen, H., ed., *GPU Gems 3*. Addison-Wesley Professional. 409–428.

[27] Policarpo, F.; Oliveira, M. M.; and Comba, J. a. L. D. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, 155–162. New York, NY, USA: ACM.

[28] Reeves, W. T.; Salesin, D. H.; and Cook, R. L. 1987. Rendering antialiased shadows with depth maps. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 283–291. New York, NY, USA: ACM.

[29] Risser, E.; Shah, M.; and Pattanaik, S. 2005. Interval mapping. Technical report, University of Central Florida.

[30] Schwarz, M., and Stamminger, M. 2007. Bitmask soft shadows. *Computer Graphics Forum* 26(3):515–524.

[31] Shah, M. A., and Konttinen, J. 2007. Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics* 13(2):272–280.

[32] Soler, C., and Sillion, F. X. 1998. Fast calculation of soft shadow textures using convolution. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 321–332. New York, NY, USA: ACM.

[33] Tevs, A.; Ihrke, I.; and Seidel, H.-P. 2008. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 183–190. New York, NY, USA: ACM.

[34] Tzeng, S., and Wei, L.-Y. 2008. Parallel white noise generation on a GPU via cryptographic hash. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 79–87. New York, NY, USA: ACM.

[35] Williams, L. 1978. Casting curved shadows on curved surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, 270–274. New York, NY, USA: ACM.

[36] Woo, A.; Poulin, P.; and Fournier, A. 1990. A survey of shadow algorithms. *IEEE Comput. Graph. Appl.* 10(6):13–32.

[37] Wyman, C., and Hansen, C. 2003. Penumbra maps: approximate soft shadows in real-time. In *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering*, 202–207. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association.

[38] Wyman, C. 2005. Interactive image-space refraction of nearby geometry. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, 205–211. New York, NY, USA: ACM.