EFFICIENT MULTIFRAGMENT EFFECTS ON GRAPHICS PROCESSING UNITS

by

Louis Frederic Bavoil

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

May 2007

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Louis Frederic Bavoil

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Cláudio T. Silva
	João L.D. Comba
	Peter Shirley

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Louis Frederic Bavoil in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Cláudio T. Silva
Chair: Supervisory Committee

Approved for the Major Department

Gary Lindstrom
Chair/Director

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Current GPUs (Graphics Processing Units) are very efficient at rendering opaque surfaces with local lighting based on the positions of point lights and the nearest fragment to the eye for every pixel. However, global illumination features such as shadows and transparent surfaces require more fragments per pixel. Multifragment effects on the GPU usually require multiple geometry passes which are expensive for large scenes.

In the first section of this thesis, we look at how to capture rasterized fragments efficiently on current GPUs. First, we analyze how to feed the GPU geometry by studying the impact of batch size and vertex attributes on performance. Then, we describe a data structure that we refer to as the *k*-buffer, which can be implemented on current GPUs with programmable read-modify-write operations on multiple fragments per pixel. The *k*-buffer can be used to perform depth peeling in a single geometry pass, but its current GPU implementation suffers from pipeline hazards. We propose two ways of removing these hazards in future GPUs.

In the second section, we present effects using multiple fragments per pixel. First, we present a robust soft shadow mapping algorithm based on multiple layers of fragments per shadow map pixel. Similarly to related work, our algorithm renders soft shadows from rectangular lights, based on the idea of unprojecting the shadow map pixels into the world, and using them as a simplified geometry of the occluders. The algorithm runs interactively on the GPU and addresses the issues of self-shadowing and light bleeding robustly, using multiple depths per shadow map pixel. Finally, we look at other effects that can take advantage of the *k*-buffer: transparency, translucency, constructive solid geometry and depth of field.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would probably never have done research in visualization and computer graphics without Cláudio T Silva, who gave me the opportunity to work for him in my spare time when I met him at the Oregon Graduate Institute in 2003. He then offered me an internship at the Scientific Computing and Imaging (SCI) Institute, University of Utah, in Spring 2004, and encouraged me to apply to graduate school. Cláudio has been a great advisor. He taught me how to do research in graphics, and kept feeding me with interesting ideas and comments. I also owe him a two-week visit in João Comba's group at UFRGS (Universidade Federal do Rio Grande do Sul) in Brazil, and an internship opportunity at Sony Playstation Research and Development, where I spent 6 months in 2005. During my 3 years in Cláudio Silva's group at the SCI Institute, I had the chance to work on a variety of topics, mostly with people from University of Utah, but also with some external collaborators from national laboratories and companies:

- Simplification of tetrahedral meshes by point sampling [70] with Dirce Uesu, Shachar Fleishman, Jason Shepherd and Cláudio Silva,
- Out-of-core rendering of large meshes based on the iWalk system from Wagner T Correa (now at IBM) and Cláudio Silva,
- The VisTrails visualization system [7], with Steven Callahan, Patricia Crossno (at Sandia National Labs), Juliana Freire, Carlos Scheidegger, Cláudio Silva, and Huy Vo,
- A client-server progressive visualization system for unstructured grids [11] with Steven Callahan, Valerio Pascucci (at LLNL) and Cláudio Silva,
- Soft shadow mapping on the GPU [6] with Steven Callahan and Cláudio Silva,
- And multifragment effects on the GPU using the *k*-Buffer [5] with Steven Callahan, João Comba, Aaron Lefohn (at Neoptica), and Cláudio Silva.

I thank Sony Computer Entertainment America (Sony Playstation), for having accepted me in their graphics research and development group, for an internship under the supervision of Alan Heirich. Alan and I developed together a gem of soft shadow mapping algorithm for

GPUs, which I improved at Utah and turned into a SIGGRAPH sketch [8] in 2006, a technical report [6], and finally a chapter of this thesis. I thank Gabor Nagy from Sony, who shared shadow-mapping code with me from his free modeling and rendering software for Linux called Equinox-3D, which he has been developing for 10 years or so. I also thank Antoine Labour and Axel Mamode, who educated me about GPU performance during my internship at Sony. I thank Gael Guennebaud, from IRIT in Toulouse, France, and Eric Paquette, from LESIA in Montreal, Canada, for influential discussions at SIGGRAPH, which helped me improve my soft shadow mapping technique. In particular, Gael Guennebaud was kind enough to send me code to help me implement his algorithm [27]. Steven Callahan has always been of great help in discussing and presenting ideas.

I thank João Comba from UFRGS in Brazil, who took care of me there for two week in Spring 2005, and supervised me as I was working on image-based reprojection algorithms for GPUs with one of his students, Carlos Dietrich. This project gave me the idea of un-projecting a shadow map, on which I built my shadow mapping work a few months later at Sony. Discussions with João were always very helpful for clarifying my ideas. I thank Peter Shirley for his enthusiasm, and his fun class on realistic rendering who inspired me to render translucency with Fresnel's law. I also thank my other teachers in computer graphics: Cláudio Silva, Emil Praun, Steven Parker, and Elaine Cohen. I thank Aaron Lefohn, from Neoptica, for discussions about graphics hardware, his challenging comments, and sharing 3D models with me. I thank Milan Ikits and Steven Callahan for having implemented the first *k*-buffer on GPU [12], and Erik Anderson for the first port of a *k*-buffer to GLSL, on which I built my *k*-buffer applications. I also thank Milan Ikits and Carlos Scheidegger for discussions about potential *k*-buffer applications. I thank Tilo Ochotta, from University of Konstanz, Germany, for discussions about high performance rendering of triangle meshes on the GPU.

I thank Mathias Schott for pointing me to original 3D models, Antony Romrell from the College of Fine Arts at University of Utah for the spider model, Gabor Nagy for the table scene, `3dplants.com` for the weed model, `planit3d.com` for the thin tree model, Taylor Holliday at UC Davis for the thick tree model, and `scifi3d.theforce.net` for the AT-AT model. I thank NVIDIA for donated hardware, and ATI for beta hardware. This work has been supported by the National Science Foundation under grants CCF-0401498, EIA-0323604, OISE-0405402, IIS-0513692, CCF-0528201, and OCE-0424602, the Department of Energy, an IBM Faculty Award, and a University of Utah Seed Grant.

CHAPTER 1

INTRODUCTION

Graphics Processing Units (GPUs) have a very different processing pipeline from general Central Processing Units (CPUs). As CPUs and GPUs evolve and become more alike, GPUs are still particularly fast for raster-based graphics because of their dedicated hardware components: a rasterizer transforming geometry primitives into an image, vertex and texture caches, and read-modify-write raster operations.

1.1 GPU Background

The classical CPU pipeline can be partitioned into five macro stages: instruction fetch, instruction decode, execution, memory access, and write back. The GPU pipeline has the following stages: vertex fetch, vertex shading, rasterization, fragment shading, and raster operations. At a given time, only one vertex or fragment shader can be active in the GPU pipeline. Vertex shaders typically transform vertex positions and perform animation, while fragment shaders perform pixel-level lighting and shadowing.

Originally, GPUs were not programmable. The geometry to be rendered, and the various parameters of the graphics pipeline were specified using fixed functions (either in OpenGL or in DirectX). In 2001, with the NVIDIA GeForce 3 and the ATI Radeon 8500, vertex shaders became programmable, either in a specific assembly language, or in higher-level languages such as Cg, or HLSL in DirectX 8. Shaders were typically written in assembly language with dedicated instruction sets. In 2002, fragment shaders became programmable as well, but with many limitations. For example, fragment shaders could be only 96 assembly instructions long, and had no dynamic flow control. In 2002, the Radeon 9700 introduced a feature called Multiple Render Targets (MRTs). This feature extends the size of the output of a fragment from 1 RGBA color to k RGBA colors. The most common use of MRTs is to store the color, normals and texture coordinates of the nearest visible fragments into multiple textures, and shade them once in a postprocessing pass, a technique called deferred shading. In 2003, floating point textures enabled high dynamic range environment maps, along

with more general-purpose computation on the GPU (GPGPU). In 2005, with the GeForce 6, fragment shaders were extended to support dynamic flow control, enabling conditional branches, function calls, and dynamic loops. At the same time, the maximum number of instructions of shaders jumped from 96 to 65k. Floating-point textures were also supported, along with 32-bit shader arithmetics. In 2006, among other things, the GeForce 8 GTX has increased the total number of fragment pipelines from 24 four-way SIMD processors to 128 scalar processors, and the number of supported multiple render targets from 4 to 8.

1.2 General Motivation

With all this fragment shading power available since 2002, the research in GPU algorithms has concentrated on shifting more work to the fragment shaders. For instance, there has been a resurgence of research in real-time soft shadow mapping algorithms, with three algorithms based on backprojection published in 2006 from Atty et al. [2], Guennebaud et al. [27], and Aszdi and Szirmay-Kalos [3], and one algorithm based on cone culling by Bavoil and Silva [8]. The common motivation of these algorithms, and of the multifragment effects presented in this thesis, is to achieve an effect that would have required multiple passes over the geometry, with less rendering passes, and more fragment shader work. In the case of soft shadows from an area light, an approach which can converge to a physically-correct image is to sample the area light with many point lights and average the resulting hard shadows [32]. This algorithm requires more than 100 passes to make banding artifacts disappear, which are unrealistic discontinuities in the shadow gradients. Although this algorithm works for small scenes (assuming that the shadow map resolution is sufficient), it is too slow to be interactive on scenes containing millions of triangles. In this thesis, we follow the approach of reducing the number of geometry passes at the cost of more fragment shading work, for the case of soft shadow mapping, and multifragment effects in general.

1.3 Multifragment Effects

Multifragment effects are effects that require access to more than one fragment per pixel to compute the color of this pixel. Examples include transparency, translucency, constructive solid geometry, depth of field, and shadow mapping. In this introduction, we focus on two effects: transparency (e.g., Figure 1.1) and shadow mapping (e.g., Figure 1.2).

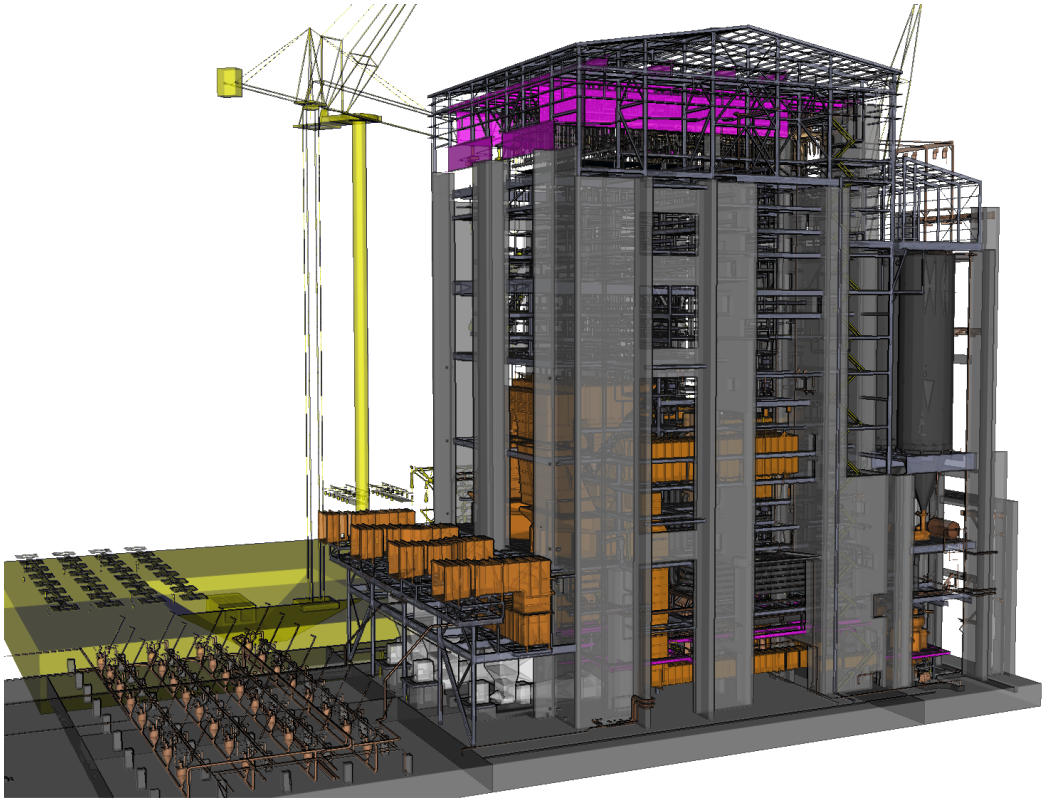


Figure 1.1. Transparent Powerplant model, with four color layers captured using our single–pass depth peeling implementation.

1.3.1 Transparency

To render transparent surfaces exactly with GPUs, all the rasterized fragments corresponding to a given pixel must be composited, either front to back or back to front. The opacity of the fragments is traditionally stored in the α channel of the RGBA fragments. This opacity can be specified with the geometry, and later modified by the fragment shader to take into account Fresnel's effect [42]. Assuming no intersecting primitives and no visibility cycles, the most common way of rendering transparency is to render the primitives in back to front order using an object-space algorithm on the CPU, and to blend the fragments with the color buffer using alpha blending. However, ordering the primitives may be too expensive for large scenes, and in case of visibility cycles, such an order may not exist.

The burden of sorting the geometry can be shifted from the CPU to the GPU using *depth peeling* [51, 22]. Depth peeling is an image-based technique that uses a depth buffer to capture not only the first visible layer of fragments from the eye, but also all the underneath layers of fragments. On current GPUs, each layer requires to rasterize the whole geometry an additional time. We call each of these steps a geometry pass.

(a) (b)

Figure 1.2. Image rendered with and without shadows. The soft shadows were rendered using our algorithm with three shadow-map layers and 961 samples per pixel.

1.3.2 Shadow Mapping

Shadows in GPU applications are typically handled using shadow mapping [75, 64]. A shadow map is a depth image rendered from the point of view of the light. Every pixel on the screen finds a corresponding fragment in the shadow map to determine if it is in shadow or not. Thus, as any effect requiring other geometry fragments to be looked up at every pixel, shadow mapping is a multifragment effect. One intrinsic problem with shadow mapping is the instability of the binary shadow test, which compares the depth of the current shading point with the depth of the corresponding shadow map pixel. When the two compared depths correspond to the same surface, this test is unstable because of the limited resolution of the shadow map, and potential quantization issues. A robust solution to this problem is midpoint shadow mapping [79], which can be performed by depth peeling the first two depth layers from the light, and using the average depths as a shadow map. This is another example of multifragment algorithm, requiring three fragments per pixel (two shadow map layers, and the eye depth), and therefore three geometry passes on current GPUs.

Chapter 5 describes a soft shadow mapping algorithm based on multiple shadow map layers to handle self-shadowing robustly, trying to minimize the number of parameters. The algorithm handles light bleeding due to gaps between shadow map pixels by using a layered depth image, similarly to Agrawala et al. [1]. The occlusion values of each shadow map fragment are computed by backprojecting the fragment onto the light plane like in the algorithm from Guennebaud et al. [27]. See Figure 1.2 rendered with our soft shadow algorithm.

1.4 Contributions

In this thesis, we build mainly on the work of Everitt [22] and Callahan et al. [12], for capturing multiple fragments per pixel, and on the work of Woo [79], Agrawala et al. [1], Im et al. [39], and Guennebaud et al. [27] for our soft shadow mapping algorithm. The main contributions of this thesis include:

- We analyze in depth optimized ways to feed current GPUs with geometry. In particular, we provide benchmarks for computing the optimal batch size and attribute types for different GPUs.

- We propose two ways to modify the current GPU pipeline to enable capturing multiple fragments per pixel in a single pass using a programmable *k*-buffer on future GPUs. We show multifragment effects that would be rendered more efficiently using the *k*-buffer. We demonstrate the feasibility of the *k*-buffer, with a hardware implementation on current GPUs prone to pipeline hazards, and show heuristics to minimize these hazards.

- We present a novel soft shadow mapping algorithm based on a layered shadow map. Because this algorithm captures multiple depths per shadow map pixel, it handles self-shadowing and light bleeding more robustly. Since our algorithm is physically-based, it has only a few quality parameters, which are the resolution of the shadow map, the number of shadow-map layers, and a maximum number of samples per pixel. Degenerate self-shadowing cases can be handled with a slope-based bias with one or two additional parameters.

The remainder of this thesis is organized as follows. In Chapter 2, we provide background information on rendering geometry on GPUs. In Chapter 3, we benchmark the geometry processing by current GPUs and derive an optimal way of passing geometry to the GPU. In Chapter 4 we present the *k*-buffer, and its use for single-pass depth peeling. In Chapter 5, we describe our robust soft shadow mapping algorithm. In Chapter 6, we describe multifragment algorithms that can be performed more efficiently using the *k*-buffer. Finally, in Chapter 7, we conclude and discuss potential future work.

CHAPTER 2

BACKGROUND

2.1 The GPU Pipeline

As of 2006, GPUs render images from 3D geometric primitives (points, line segments or triangles) by transforming the primitives into small squares of the size of a pixel (or subpixels for antialiasing) called fragments. This process is called *rasterization*. An advantage of rasterization over ray tracing [66] is that primitives can be rasterized independently of one another, which enables efficient parallel implementations with local memory access.

2.1.1 Overview

Figure 2.1 gives a succinct overview of the stages of the current GPU pipeline, without antialiasing. For more information, refer to the "Real-Time Rendering" Book [56], the GPU Gems 2 article about the GeForce 6 Architecture [45], and SIGGRAPH tutorials such as [67]. Before rasterization, the vertices of the primitives go through a *vertex shader*, which projects their position and optionally defines vertex attributes. The vertices are then assembled into primitives, which are rasterized. The vertex attributes are linearly interpolated by the rasterizer inside each primitive. The rasterized fragments are shaded using a *fragment shader* which assigns to them one or more color values, and optionally a depth value. Fragment shaders compute their output based on fragment attributes and texture fetches. Vertex shaders can also read from textures, but with more latency. In 2006, with DirectX 10, a new stage appeared in the pipeline right after the vertex shaders, called geometry shaders. They can be used to generate primitives (e.g., for growing hair), and optionally to stream out primitives to geometry memory without going through the rasterizer. Since multiple fragments (or fragment samples with full-scene antialiasing) may occupy the same image pixel, another stage of the pipeline is necessary to create an image from multiple shaded fragments per pixel. The operations of this stage are called the *raster operations*, or *output merger* in the DirectX 10 terminology [10].

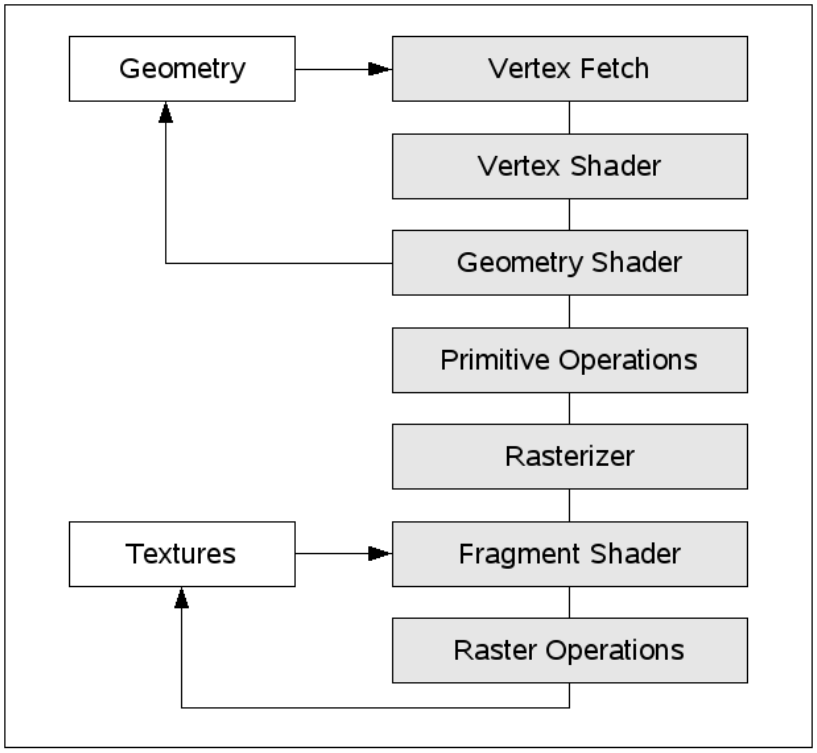


Figure 2.1. A simplified view of the DirectX 10 GPU pipeline.

2.1.2 Raster Operations

The raster operations are the only read-modify-write operations in the GPU pipeline. For efficiency reasons, they are nonprogrammable. The only programmable stages of the pipeline are the vertex shader and the fragment shader. For opaque surface, the first visible fragment for each pixel is selected using a Z-Buffer [15]. A Z-Buffer is a 2D read-modify-write buffer of depth values, which stores the current minimum or maximum depth value for every pixel. To find the nearest fragment to the eye, the Z-Buffer is initialized with the maximum possible depth value. For every rasterized fragment, if the rasterized depth is greater than the depth stored in the Z-Buffer, the fragment is discarded. Otherwise the depth value is updated and the fragment is sent to the next stage. For semitransparent surfaces, fragments can be composited with the current color buffer using blending and logical operations. These operations are functions of the RGBA color(s) of the rasterized fragment and the RGBA color(s) stored in the current color buffer containing the image(s) in progress. Blending operates on the RGB channels and the α channel, whereas logical operations operate on the bits of all the RGBA channels. In the current GPU pipeline, these two operations are the only read-modify-write operations available on the color buffer.

2.2 Graphics Programming Interfaces

There are two mainstream APIs to access the GPU: OpenGL, which is an open standard, and DirectX, a proprietary library. The OpenGL 1.0 specification was officially released in 1992 by SGI, and has been maintained by the OpenGL Architecture Review Board (ARB) until 2006. The latest version of OpenGL (2.1) is now maintained by the Khronos Group. OpenGL is a standard, and each hardware vendor provides its implementation in their drivers. Currently, OpenGL is supported by ATI (now AMD) and NVIDIA, under Linux and Windows, and by Apple, under MacOS X. OpenGL has an extension mechanism which makes it easy to add new features as the hardware evolves. For example, the GeForce 8 added 22 OpenGL extensions to OpenGL 2.1. OpenGL ES, developed by the Khronos Group, is a streamlined subset of OpenGL designed to be as close as possible to the actual hardware, while still conforming to OpenGL. The advantage of OpenGL ES are smaller drivers, and a smaller and cleaner API. In general, OpenGL features that are available in OpenGL but not in OpenGL ES are features that may not be directly hardware accelerated, but emulated by the driver.

Direct3D, the 3D API of DirectX from Microsoft was first released in 1995. Unlike OpenGL, Direct3D is a proprietary library, for which only Windows binaries are available. DirectX 10 [10], the latest version is only available for Windows Vista. With DirectX 9, DirectX had the advantage of supporting all the latest features of GPUs such as render-to-texture, while OpenGL was taking a long time to create an equivalent standard. However, OpenGL implementations now keep up with DirectX. In general OpenGL is mostly used for professional applications, while DirectX is mostly used for PC games. In the embedded world, especially the console world, the market is shared. For instance, the Sony Playstation 3 is programmed in OpenGL ES while Microsoft's XBox 360 uses DirectX.

2.3 OpenGL Rendering Modes

In this chapter and the following chapter, we focus on how to render geometry efficiently with OpenGL. One issue with OpenGL nowadays are the numerous ways to feed it geometry, also called rendering modes, or drawing modes. In this section, we present the evolution of these rendering modes.

OpenGL has introduced a way to use the GPU as a stream processor of geometry called immediate mode [80], also known as `glBegin/glEnd`. In this mode, geometric primitives to render are specified with one function call per vertex. To cache geometry on the GPU, OpenGL provides display lists. Display lists reduce the API overhead, making it possible to

render one batch of primitives with a single function call. Such a function call is also known as a draw call. Another advantage of display lists is that the geometry can be cached on the GPU, thus reducing the data traffic from system main memory to video memory. However, the time and memory overhead for compiling display lists can be an issue.

In 1995, OpenGL was extended to enable a large amount of geometry to be streamed in a single draw call, called vertex arrays. With this extension, the vertex geometry is specified as array(s) of vertex data, and the primitives are specified as an array of indices into the vertex array(s). This extension reduces the amount of API overhead required for rendering batches of dynamic geometry. The main advantage of vertex arrays is that they do not require any compilation. For static geometry, display lists of vertex arrays are efficient.

In 2003, another OpenGL extension called vertex buffer objects (VBOs), was standardized. VBOs extend vertex arrays such that vertices as well as indices can now be cached on the GPU without using display lists. Contrary to display lists, the data can now be passed directly to the GPU without any compilation overhead. However, since the application now has a direct access to the geometry data that are sent to the GPU, it is now responsible for formatting this data in a GPU-friendly way. Chapter 3 determines factors that affect the performance at the VBO level.

For scenes that are made of identical geometric objects with individual positions and attributes, instancing can be used to specify a set of primitives to be drawn n times on the hardware, with an unique instance *id* per iteration. This feature has been available in DirectX since 2003, and has been exposed in OpenGL through the `GL_EXT_draw_instanced` extension in 2006. At the same time, with the geometry shaders [10], it has become possible to stream out the outputs of a vertex shader directly into a buffer, thus shortcutting the rasterizer. The vertices written in the stream-output buffer may be redrawn in subsequent passes with no CPU intervention. While this feature is attractive, it is difficult to implement efficiently because of the feedback loop that it creates in the pipeline.

Draw calls are the API calls that activate the GPU pipeline, sending a batch of geometry to render. A draw call can either send a list of vertices without indices (nonindexed primitives), or a list of vertices and indices (indexed primitives). Issuing too many draw calls per frame is often a bottleneck with current GPUs [78, 37]. The reasons are the CPU overhead in the driver, and the GPU overhead of processing a new command. The solution is to batch primitives (e.g., triangles) before rendering so that each draw call renders a larger amount of primitives.

2.4 Transferring Memory to the GPU

When rendering large amounts of geometry on the GPU (millions of vertices), two stages of the GPU pipeline tend to become the main bottleneck: bus transfers and vertex/index fetches. The efficiency of both of these stages is directly related to how the application is feeding the pipeline with geometry.

2.4.1 Video Bus

In modern PCs, the video card containing the GPU is connected to the I/O bus through a PCI-Express (PCI-E) 16x bus [45]. The theoretical bandwidth of this bus is 4 GB/s for reads and writes. For a case with 32 bytes per vertex and no vertex indices, this means a peak performance of 128M vertices per second, or 4.2M vertices per frame at 30 fps. Still, it is always faster to avoid bus transfers when possible by caching static geometry on the video memory of the GPU, using vertex buffer objects. Also, note that in practice, due to protocol overhead and buffering issues, the actual maximum bandwitdh is less than 4 GB/s.

2.4.2 Vertex Caches

Figure 2.2 shows the stages of the memory pipeline when the GPU fetches a vertex. Vertex coordinates and attributes are transferred from the video memory of the GPU (typically DDR memory) to the GPU itself. The transfer time per vertex depends on the data types of the vertex attributes—nonnative types require conversions, and on the efficiency of the pretransform cache. There are two caches in current GPUs: a cache between the video memory and the vertex shaders, called pretransform cache, and a smaller cache after the vertex shader, called post-transform cache. On the XBOX GPU (similar to a GeForce 3), the pretransform cache contains 4KB of storage [56], and the post-transform cache can contain three or more shaded vertices depending on the vertex size [56]. More recent GPUs have a much larger pretransform cache as shown in Chapter 3.

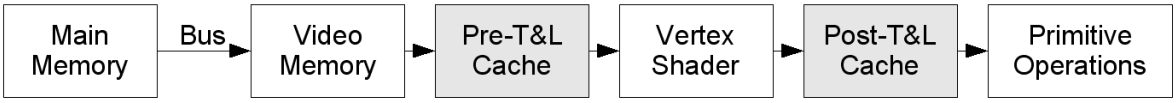


Figure 2.2. Memory stages and vertex caches.

CHAPTER 3

OPTIMIZING RAW GEOMETRY RENDERING

The goal of this chapter is to find a set of rules that give best performance for both ATI and NVIDIA hardware when rendering large amounts of geometry on a GPU. The geometry can be static in which case the vertex data are never updated, or dynamic otherwise. In this chapter, we focus on the static case which is the most common. Static geometry does not mean that the geometry cannot move in the application since vertices can be animated in vertex shaders.

3.1 Introduction

As introduced in Chapter 2, there are many ways to feed GPUs with geometry. With OpenGL, one can use immediate mode, vertex arrays, display lists and vertex buffer objects (VBOs). With DirectX, there are only vertex buffers, which are an equivalent of VBOs. GPU vendors recommend always using VBOs. Since OpenGL ES supports only VBOs and vertex arrays, we think that immediate mode and display lists are just abstractions of the driver which compile into VBOs. However, display lists may be faster than VBOs with certain pathological VBO configurations as we show in Section 3.3. Besides rendering modes, the precision in which the vertex positions and attributes are specified impacts memory transfers from main memory to video memory, and from video memory to the GPU vertex processors. There are many ways to specify vertex positions and attributes using various data types such as bytes, short integers (16 bits), or floating-point numbers (32 bits). Besides, in many cases, the geometry can be grouped into batches of primitives and the vertex positions and normals in each batch can be quantized, decompressing the vertex data on the GPU. Simple quantization according to bounding boxes, and decompression in vertex shaders can speed up vertex transfers substantially [13].

This chapter also studies the impact of indexed primitives and batching. In particular, we study the efficiency of batches of indexed primitives according to batch size and discover that for the GPUs we have tested, vertex processing performance depends not only on the number of batches and the total number of vertices, but also on the efficiency of the pretransform

vertex cache. We are interested in deriving the common parameters that achieve best VBO performance on ATI and NVIDIA graphics hardware. The GPU architecture for fetching vertex geometry, which we study in this section, has been the same from at least the GeForce 3 generation to the GeForce 7 generation, and we believe that it will remain similar, as long as GPUs will be based on rasterization. The main contributions of this chapter include an in-depth study of a large number of ways to pass geometry to a GPU for various dataset sizes and data layouts, and a performance comparison of NVIDIA and ATI recent hardware, enabling developers to design their applications to perform well with a broad range of GPUs.

3.2 Benchmarks

We developed two benchmarks for our experiments: a procedural benchmark cloning blades of grass, which allows us to easily generate scene with arbitrary number of triangles, rendered in batches, and a simpler benchmark which can render multiple triangle meshes. In both benchmarks, the fragment shader outputs a constant color and the depth buffer is enabled with the default depth test. We made sure that the GPU pipeline was not limited by rasterization or any later stage by checking that resizing the window did not have any effect on frame rate. Note that the results from the grass benchmark can be reproduced using a simpler benchmark that renders visible points with random positions. Our GeForce 7800/7900 GTX setup used driver 1.0-8776 under Linux, our Radeon X800 Pro, Catalyst 6.11 under Windows, and our GeForce 6600, Mac OS X 10.4.8. The render times were measured on the CPU using `glFinish()`, and averaged over 10 seconds.

3.3 OpenGL Rendering Modes

We evaluated the performance of multiple OpenGL rendering modes with our grass benchmark. Table 3.1 shows two interesting facts. First, VBOs can be slower than other rendering modes. Second, display lists of vertex arrays can be slower than basic vertex arrays.

In these results, the immediate mode uses `glBegin/glEnd`, the display list mode renders a display list of a vertex array, the vertex array mode does not use a display list, and the the VBO + EBO mode uses one VBO for the vertex positions, one VBO for the vertex normals, and one element buffer object (EBO) for the indices. Since this experiment was rendering the whole geometry in a single draw call, the pretransform cache of the GeForce cards was overwhelmed and their VBO performance was far from optimal. In this case, when using

Table 3.1. Performance of OpenGL rendering modes with a single draw call. Grass Benchmark. Rendering 2.1M vertices (1.5M triangles) per frame, with 32-bit `GL_FLOAT` vertex positions. GPUs: NVIDIA GeForce 7800 GTX under Linux, GeForce 6600 under MacOSX, and ATI Radeon X800 Pro under Windows, and Radeon 9600 Pro under MacOSX.

Method	GeForce 7800 GTX	GeForce 6600	Radeon X800 Pro	Radeon 9600 Pro
Immediate Mode	0.80 fps	7.10 fps	7.73 fps	3.36 fps
Vertex Arrays	24.54 fps	9.45 fps	4.56 fps	7.43 fps
Display Lists	21.07 fps	8.31 fps	48.4 fps	17.5 fps
VBOs + EBO	15.54 fps	4.40 fps	90.5 fps	54.8 fps

floating-point normals, display lists and vertex arrays are faster than VBOs. In the rest of this chapter, we focus on VBOs, which are the fastest rendering mode in most cases.

3.4 Vertex Throughput

A fundamental question about how fast a GPU can draw geometry is the triangle throughput. The triangle throughput of current GPUs may depend on the number of triangles or the number of vertices per frame. Indeed, rasterization peak performance is measured in triangles per second, but vertex fetch or vertex shading are most often the bottleneck. In this experiment, we increased the total number of triangles in the scene and we rendered them with a single draw call. We plot the frame rate as a function of the number of vertices in the scene on Figure 3.1. For large vertex counts on GeForce 7, the performance drops abruptly when rendering more than 1M vertices in a single `glDrawElements` call. On Radeon X800, the driver crashed over 2M vertices per draw call.

The conclusion is that on NVIDIA's GeForce 6 and 7 series, rendering too many vertices with `glDrawElements` overwhelms the pretransform vertex cache and greatly reduces performance. Note that nonindexed primitives rendered with `glDrawArrays` do not have this problem. We focus on indexed primitives because they allow sharing vertices, which saves vertex shading when rendering meshes. The problem then becomes finding the maximum number of vertices per draw call that does not overwhelm the pretransform cache. Our experiments show that on GeForce 7, the pretransform cache can store up to 1M floating-point vertices. The solution is to split the dataset into an union of smaller datasets, called batches, and to render each of these batches of geometry with a separate draw call, a technique called batching [78].

3.5 Batching

As the two previous experiments showed, rendering the whole geometry in one batch may trash the pretransform cache and greatly hurt performance. Batching triangles means taking a soup of triangles and merging or partitioning them into groups of a given maximum size. There are multiple ways to perform batching on a large triangle mesh.

3.5.1 Stream Batching

The simplest way is to sequentially break the stream of triangles while not reordering the triangles. This works well assuming that the original triangles have good locality. Each blade in our grass benchmark is made of five adjacent triangles. These blades are accumulated in batches of consistent locality, where locality can be defined from positions, normals or topology. We ran our grass benchmark with various numbers of batches and a fixed total number of triangles in the scene. Figure 3.2 shows that the optimal number of batches per frame is a single batch for the ATI Radeon X800, whereas for the NVIDIA GeForce 7, the sweet spot is between 500 and 1,000 batches per frame. Note that for very large datasets though (i.e., `lucy`, 28M triangles), the ATI driver crashed. So batching is also necessary with ATI hardware for very large datasets.

3.5.2 Cache-Coherent Batching

Stream batching works well for models in which the maximum Euclidian distance between successive triangles is small, which is the case for most models generated by modeling packages. However, in certain cases such as the Stanford Bunny dataset, stream batching would return nonmanifold batches of triangles, which reduces the efficiency of the vertex cache and early-z culling. Given a triangle mesh with triangle primitives, one can reorder the primitives to improve the locality of the mesh and therefore cache efficiency.

Ochotta and Hiller use principal component analysis (PCA) and split across the major principal axis to create patches from a point set [60]. The patches are split recursively until each patch conforms to a vertex budget (maximum patch size). Nehab et al. [58] use normals to drive their greedy subdivision algorithm. Yoon et al. developed a cache-oblivious triangle reordering method based on graph partitioning [81], which has an open source implementation called OpenCCL. Version 1.2 of the library works well for manifold meshes. However, in our experiments, it did not handle nonmanifold meshes with disconnected parts conservatively: some triangles were silently skipped and did not appear in the output.

3.6 Optimizing Indices

To maximize cache hits in the post-transform cache, the locality of the vertices must be improved. The goal of the post-transform cache is to avoid fetching and shading a shared vertex multiple times. There are two ways to help the post-transform cache at the application level: reordering the primitives, or using compressed primitives such as triangle strips or triangle fans.

3.6.1 Index Precision

When the number of vertices in a given batch is less than the maximum integer represented using 16 bits, `GL_UNSIGNED_SHORT` indices are faster than `GL_UNSIGNED_INT` indices. The drawbacks with short indices are that small batches of less than 65,536 vertices are required and batching with good locality may be difficult to perform. See Table 3.2.

3.6.2 Triangle Strips

For meshes, another way to optimize for the post-transform cache which takes advantage of common edges between adjacent triangles is to not use triangle primitives, but rather use triangle strips or triangle fans. Hoppe introduced a greedy algorithm for growing triangle strips [34]. This algorithm has become part of the D3DX library of Microsoft DirectX. Another greedy algorithm was implemented by NVIDIA in the NvTriStip open-source library. It works well for manifold meshes and can optimize the strips for a given size of post-transform cache.

3.7 Optimizing Vertex Data

VBOs may pass the vertex data directly to the hardware, without conversion in the driver or in the hardware. Therefore, nonnative data types may go through a slower path than native data types. In this section, we study the impact of the data types of vertex attributes on the performance of vertex processing.

Table 3.2. Performance with and without OpenCCL, with 16-bit and 32-bit indices. Dataset: Stanford Happy Buddha (543,652 vertices, 1,087,716 triangles, 64k tris/batch).

Layout	Indices	GeForce 7800 GTX	Radeon X800 Pro
Default	32 bits	110 fps	97.4 fps
Default	16 bits	119 fps	107 fps
OpenCCL	32 bits	131 fps	116 fps
OpenCCL	16 bits	144 fps	127 fps

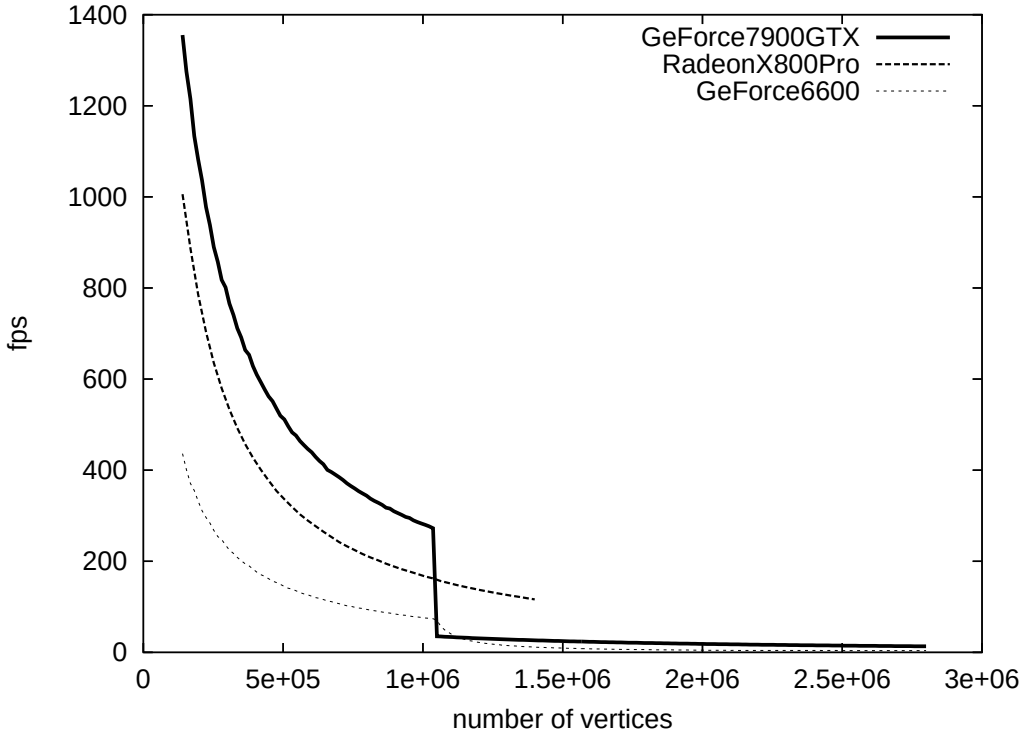


Figure 3.1. Vertex throughput. Rendering an increasing number of vertices, with a single draw call and no vertex attributes, using the grass benchmark.

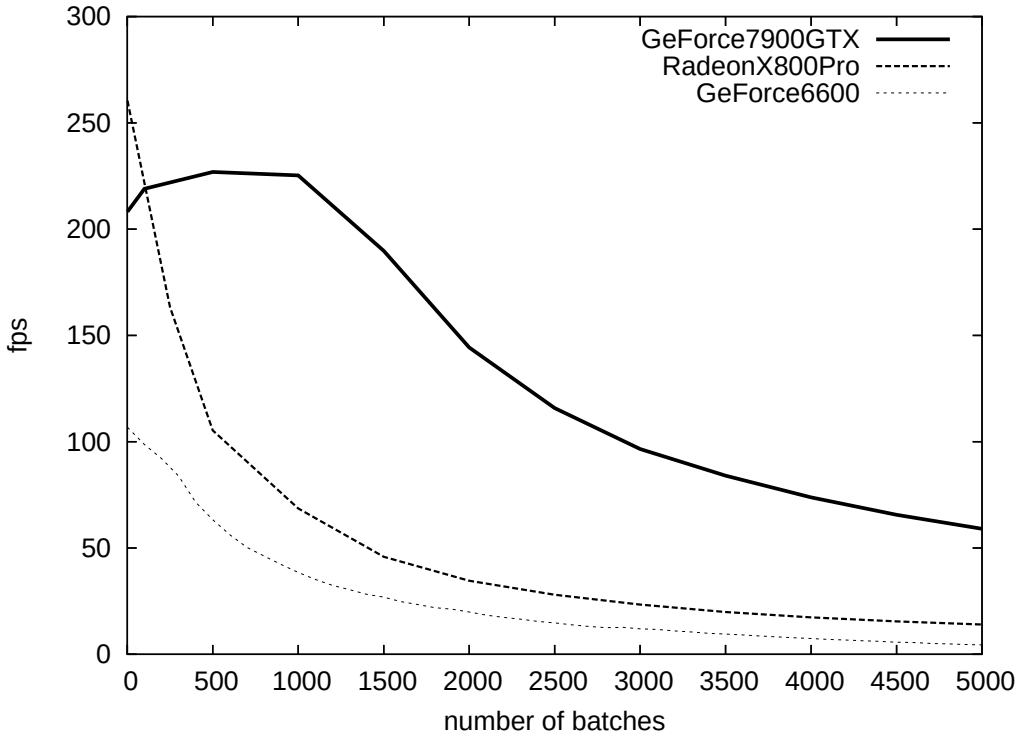


Figure 3.2. Batching. Rendering 500,000 total triangles with different number of batches and no vertex attributes.

3.7.1 Vertex Positions

Quantization can be used to encode vertex positions with less precision, for instance encoding an offset relative to a bounding box. Deering, who pioneered geometry compression [17], analyzed the required precision for vertex positions, colors and normals. Positions do not need to be floating points because the exponent part of the positions is controlled by the modelview matrix. Deering found that 16-bit short integers were sufficient. Calver proposed simple quantization strategies for which a vertex shader can do the decompression [13]. We implemented a simple quantization scheme that converts floating-point coordinates to short integers. The coordinates are first shifted and scaled to $[-1,1]^3$ according to the bounding box of the batch, and then converted to short (16-bit) integers. Table 3.3 shows that on NVIDIA hardware, using `GL_SHORT` instead of `GL_FLOAT` vertex data wins, but on ATI hardware, using `GL_SHORT` is very slow.

3.7.2 Vertex Attributes

On current GPUs, vertex attributes can be a combinations of a normal, one or two colors and up to 8 texture coordinates. With OpenGL 2.0, generic attributes can be specified with `glVertexAttrib` functions. Still, there may be differences between attributes when specified with the traditional functions, such as `glNormal`, and `glColor`.

Assuming the normals are normalized, normals can be encoded exactly in spherical coordinates, with an extra decompression cost in a vertex shader. Deering [17] determined that an angular density of 0.01 radiance between normals were enough to not introduce visible artifacts and chose to use three 16-bit signed components per normals. Normalized normals can be represented in spherical coordinates with two components or in tangent space with two tangent coordinates. In our benchmarks, we used three components per normal with various precisions.

Table 3.3. Impact of precision of vertex positions on performance, comparing 16-bit and 32-bit precision, with and without stream batching. Datasets: Happy Buddha (1M triangles) and Thai (10M triangles), with 1M triangles per batch.

Dataset	Vertex Precision	GeForce 7800 GTX	Radeon X800 Pro
Happy	`GL_FLOAT`	158 fps	107 fps
	`GL_SHORT`	161 fps	1.01 fps
Thai	`GL_FLOAT`	10.1 fps	2.56 fps
	`GL_SHORT`	10.8 fps	0.35 fps

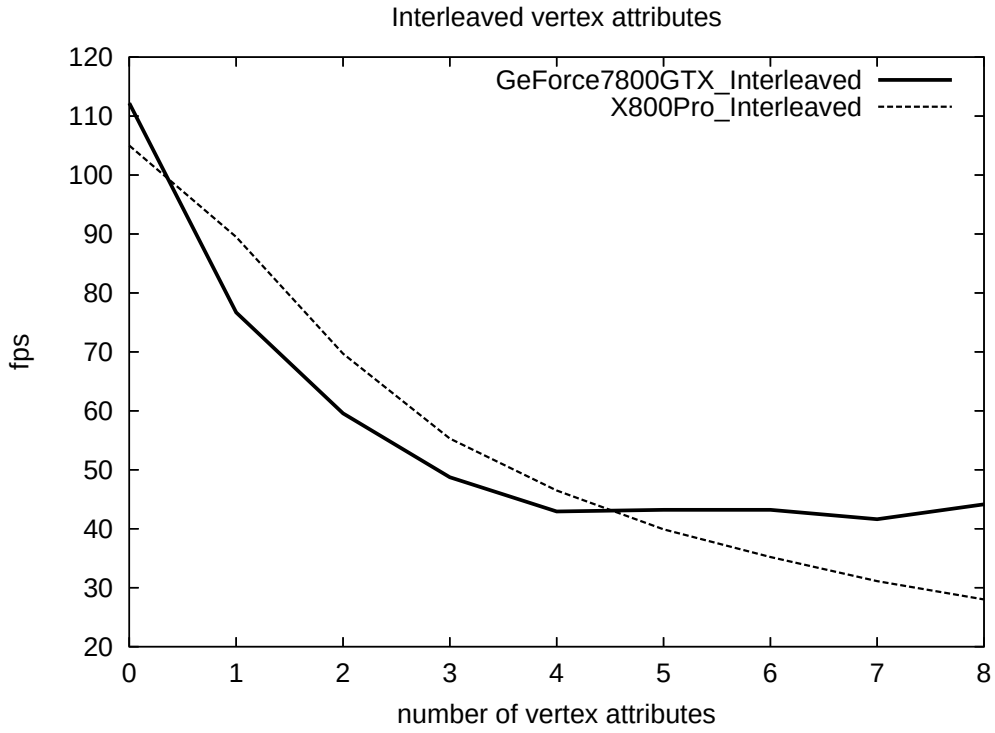
Traditionally, colors have been stored with one byte per component. Recently, with floating-point textures supported in GPUs, 32-bit floating point colors, also known as high dynamic range (HDR) colors, became available. In our benchmarks, we compared traditional byte colors with HDR colors, at the vertex fetch level.

We rendered 500,000 triangles with a single draw call and different vertex attributes. Table 3.4 shows a comparison of the frame rates according to the possible data types used for normals and colors. "3f" stands for three floating point values per attribute, "3b" for three bytes, "3s" for three short integers, and "4ub" for four unsigned bytes. We conclude that for both ATI and NVIDIA the optimal data format are three floats per normal and four unsigned bytes per color. Table 3.4 shows that normals have the same behavior as texture coordinates for shorts and floats. Note that the performance of the GeForce in this benchmark is much better than in Table 3.1 because in this case the whole VBO fits into the pretransform cache.

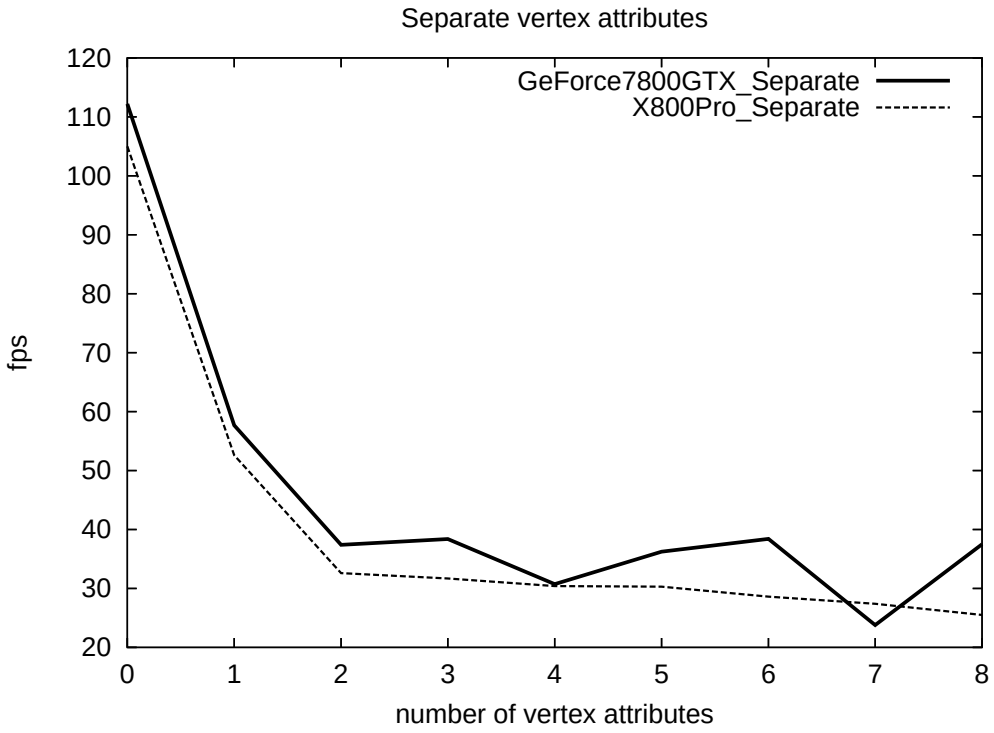
Because the vertex memory is fetched in blocks, interleaving vertex coordinates and vertex attributes is more efficient than using one vertex array per attribute. In our benchmark, we used dummy vertex and fragment shaders. Our vertex shader touched all of the texture coordinates in play to avoid unused vertex attribute fetches to be optimized out. The vertex shader reads n texture coordinates and adds them together. All raster operations were turned off to maximize performance. Figure 3.3 shows the difference between interleaved and separate vertex buffers on the Stanford Happy Buddha (1M triangles) with 64k triangles per batch and n 4-float texture coordinates. The conclusion is that interleaving is always faster than separate vertex buffers and the performance boost increases with the number of attributes.

Table 3.4. Impact of precision of vertex attributes on performance with our grass benchmark, rendering 0.5M triangles in a single VBO.

Enabled Attribute	GeForce 7800 GTX	Radeon X800
No Attribute	397 fps	259 fps
3f Normal	207 fps	256 fps
3s Normal	253 fps	5.1 fps
3b Normal	41 fps	4.9 fps
4ub Color	311 fps	257 fps
4f Color	228 fps	244 fps
3ub Color	322 fps	243 fps
3f Color	237 fps	243 fps
3f TexCoord	205 fps	193 fps
3s TexCoord	252 fps	5.4 fps



(a)



(b)

Figure 3.3. Interleaved attributes. Frame rate according to number of texture coordinates on NVIDIA GeForce 7800 GTX and ATI Radeon X800 with and without interleaved attributes. Dataset: Happy Buddha (1M triangles).

3.8 Conclusions

Section 3.3 showed that other rendering modes can be faster than vertex buffer object (VBOs) in pathological cases where the VBOs do not fit in the pretransform cache. For this reason, batching primitives such that each batch fits inside the pretransform cache may be necessary. Section 3.4 showed that the size of the vertex buffer objects can have a big impact on performance. The safest approach to set the maximum batch size is to use vertex buffers that can be addressed with short indices (at most 65,536 vertices per batch). To improve cache efficiency, one can either use indexed primitives ordered with a cache-coherent mesh layout, or use triangle strips, or a combination of both: triangle strips from a cache-coherent layout.

As Section 3.7 showed, certain GPUs do not support `GL_SHORT` vertex data, in which case they use a slower path. When building vertex buffer objects (or vertex buffers in DirectX), the safest way is to use `GL_FLOAT` for vertex positions and all the vertex attributes, even though the precision is higher than necessary. For colors, however, unsigned bytes are slightly faster than floats. Section 3.7 also showed that it is substantially faster to use interleaved vertex data, rather than separate vertex buffers.

We believe that the big difference in performance between the various ways of feeding geometry to a GPU makes it necessary for research papers to be more explicit about implementation details and bottlenecks. Otherwise, performance results are much harder to reproduce, and algorithms more difficult to evaluate. Moreover, it is important to know the response curves of certain GPUs when designing a 3D engine for generic GPUs. For example, large scenes should be batched as a preprocess during content creation.

CHAPTER 4

SINGLE-PASS DEPTH PEELING

4.1 Introduction

Raster-based graphics algorithms simulate many effects by operating on multiple fragments in the same pixel. Several existing algorithms keep all the fragments for each pixel [14, 52, 77] so that they can be sorted and composited in either front-to-back or back-to-front order for transparency. However, the unbounded memory requirements for these types of algorithms is a limiting factor for practical applications.

Recent work by Callahan et al. [12] proposed the *k*-buffer—a fixed size buffer of fragments per pixel that is maintained in GPU memory. This *k*-buffer was shown to be effective for sorting and compositing fragments in the special case of direct volume rendering with current graphics hardware. However, many applications other than direct volume rendering require access to multiple fragments simultaneously. Here, we generalize the definition of the *k*-buffer to be a pool of fragments per pixel that can be read, modified, and written at the fragment level. The benefit of the *k*-buffer is that it allows fragments to be compared, ordered, blended, and discarded in a streaming manner. Thus, many effects that normally require multiple passes over the scene geometry can instead be streamed through the *k*-buffer in one pass [47, 5].

Whereas a traditional Z-buffer-based framebuffer saves fragment results for a single depth per pixel (the front-most fragment), the *k*-buffer can save up to *k* fragments with only a small increase in memory requirements. By giving access to multiple ray intersections along a viewing ray, the additional information in a *k*-buffer provides algorithms with a more global view of the scene, in turn opening up a number of new algorithmic possibilities for raster graphics. For example, the first *k* fragments per pixel in front-to-back order can be stored in a *k*-buffer, effectively performing depth peeling in a single pass. Since the *k*-buffer supports programmable RMW operations, it can also be used to implement a Z-Buffer, a stencil buffer, or arbitrary blending.

The *k*-buffer can be implemented in current hardware using read-modify-write (RMW) operations on textures at the fragment level. Currently, this feature is allowed in hardware,

though the results are undefined. Because of the highly parallel nature of fragment processing on the GPU, there is no guarantee that artifacts will not appear from overlapping geometry in screen space. We address this issue by describing solutions that avoid the race conditions that can occur with overlapping fragments. This current implementation is used for validating our k-buffer applications, generating images, and producing experimental results of the data structure. However, with a few modifications to the current GPU pipeline, we believe that full k-buffer support is possible. We propose two such modifications that would avoid RMW hazards in future hardware. We believe that the k-buffer has important implications for interactive graphics and visualization because of the number of applications that it enables or simplifies.

4.2 Related Work

4.2.1 Single-Pass Approaches

There are many relevant publications on storing and processing multiple fragments per pixel. The traditional image-based algorithm for fragment sorting is the Z-Buffer [15]. It is a streaming algorithm—for every pixel, the fragment with lowest (or greatest) depth is kept and the others are discarded. The A-Buffer [14] is an extension of the Z-Buffer which stores all the fragments rasterized per pixel in a list, which is then sorted according to depth. Fragments that belong to the same surface and that have very close depth values are merged. This algorithm is not suitable to current graphics hardware because of its unbounded memory per pixel. The R-Buffer [77] is a variation of the A-Buffer. All the fragments for the scene are stored in a single FIFO queue in memory. Although the R-Buffer was designed for hardware implementation, storing a large number of transparent fragments is not feasible in practice. Another streaming approach for fragment processing is the Z^3 algorithm [41]. Z^3 uses a fixed number of fragments per pixel and thus requires less memory than the A-Buffer or R-Buffer. When the maximum number of fragments per pixel is reached, it selects the two closest fragments and merges them together using a set of heuristics based on pixel coverage. Of these streaming methods, only the Z-Buffer has an actual hardware implementation in current GPUs. Recently, Eisemann and Decoret [20] showed that an approximate partitioning of the scene can be performed in a single pass on the GPU by voxelizing the scene. Although this technique is very efficient for volumetric effects such as transmittance shadow mapping, it does not allow effects that require the exact locations of the fragments, such as transparency. Of these algorithms, the k-buffer is most similar to the Z^3 architecture because it stores a fixed

number of fragments per pixel. The *k*-buffer can be seen as a generalization of Z^3 where the storage and insertion of the fragments has been made programmable.

4.2.2 Multiple-Pass Approaches

Due to memory resources on graphics hardware, multipass rendering is often required to achieve many effects. The F-Buffer [52] is very similar to the R-Buffer, except that it does not sort the fragments. The F-Buffer requires semitransparent surfaces to be rendered in depth order, although it may be possible to sort an F-Buffer using a bitonic sort. Implementations of the F-Buffer that require rendering the whole geometry for every pass are available on ATI's graphics hardware [36]. The original depth peeling algorithm by Mammen [51] proposes a solution for sorting fragments by *peeling* the layers in depth order in separate passes. A hardware implementation was more recently described by Everitt [22]. Depth peeling has been used for rendering order-independent transparency [22, 74, 50], volume rendering [57, 9], collision detection [33], global illumination [28, 55], and layered shadow maps [79, 6]. Kelley et al. [44] proposed a hybrid solution that stores four RGBAZ fragments per pixel, sorted front-to-back, and handles overflow with multiple passes. In each pass, the four layers are composited into a single layer and the three remaining layers are used to capture the next fragments. A recent approach similar to depth peeling is the Vis-Sort algorithm [25] which sorts 3D primitives using occlusion queries on the GPU, assuming there are no visibility cycles and no intersecting primitives. With Vis-Sort, the number of passes required to composite transparent fragments is equal to the depth complexity of the scene. The *k*-buffer simplifies multipass approaches by allowing *k* fragments to be operated on in a single pass. Since it operates in image-space, it also avoids problems with visibility cycles and intersecting primitives.

4.3 The *k*-Buffer

The *k*-buffer is a generalization of the traditional Z-buffer-based framebuffer. Instead of restricting framebuffers to a single depth value, a single stencil value, and *n* color values, the *k*-buffer uses framebuffer memory as a RMW pool of *k* entries whose use is programmatically defined by *k*-buffer operations. In essence, the recent addition of multiple render targets (MRTs) to GPUs already allows multiple fragments to be stored, albeit in textures. We take this a step further and suggest that the programmable combination of these fragments is as important as their storage to achieve many advanced effects in a single pass. The general structure of the *k*-buffer algorithms for each fragment *f* that is rasterized is as follows:

1. **Read** the k-buffer elements for this pixel from memory. These values, along with the incoming fragment f, are now available.

2. **Modify** the k-buffer elements using f.

3. **Write** the k-buffer elements back to memory and discard f.

Many effects can be performed using different types of modify operations on the k-buffer values. Generally, these modify operations fall into two types. The first type is to use the k-buffer to accumulate up to k fragments for a postprocessing pass such as deferred shading. Examples of this type of algorithm are depth-peeling and depth-partitioning, which can be used to perform effects such as transparency, translucency, midpoint shadow mapping, constructive solid geometry, and depth-of-field. The second type is to use the k-buffer as a fragment stream processor and programmable blender. An example of this type of algorithm is fragment ordering, which can be used to perform isosurfacing, direct volume rendering, and transparency of geometry with large depth complexity.

The first type of algorithm generally requires a fixed number of fragments to perform the desired effect. The k-buffer is used as temporary storage of the most significant fragments to be used in a postprocessing pass. These fragments can be rasterized in any particular order with no change in the final result. However, the second type of algorithm generally needs to consider all fragments to achieve the desired effect. In this case, when a new fragment is inserted, a blending operation is performed and a fragment is discarded. Thus, the rasterization order of the fragments will affect the output. The k-buffer is capable of sorting a k-nearly sorted sequence (k-NSS) of fragments. Given a sequence S of fragment depths, S is a k-NSS if no depth in S is more than k positions out of place. Therefore, when fragment ordering is required, the geometry needs to be rasterized in at least a partial order so that the k-buffer can complete the ordering and blend the results all in one pass (see [12] for a more formal definition of a k-NSS). The partial ordering of the geometry occurs in object-space prior to rasterization. The extent of the ordering that is required is dependent both on the depth complexity and the available k size.

4.3.1 Future Hardware Implementation

Complete k-buffer support in hardware would enable fast single-pass effects without RMW hazards that may occur with our current implementation. Although the specifics of hardware implementations are not publicly available, we propose two possible high-level solutions

based upon available information about the current hardware pipeline. In both solutions, we implement the *k*-buffer as a set of floating-point renderable buffers, and write to these buffers using Multiple Render Targets (MRTs). The main difference between the two solutions is where we execute the *k*-buffer operations (read, modify, and write). The first solution involves changes at the fragment scheduling stage of the pipeline and the second solution involves changes at the blending stage of the pipeline. Figure 4.1 shows a simplified version of the GPU pipeline with annotations that specify the areas that require modification for our hardware proposals.

In the fragment scheduling solution, the *k*-buffer operations are implemented in fragment programs, and the reads are performed using the connection between the memory partition and the texture cache. However, this approach has issues with RMW hazards because current GPUs process multiple fragments at the same time per fragment pipeline, with multiple parallel pipelines. When an output of a fragment is computed, it is not immediately written to memory, but written to an output buffer that reorders the fragments in the order in which the primitives have been rasterized. This is necessary for the correctness of RMW raster ops such as blending and stencil buffer. This asynchronous write to memory means that if two overlapping fragments are processed concurrently, and they modify a value of the *k*-buffer for the same pixel, one fragment will read an obsolete value, and overwrite the value of the other fragment with an incorrect value. A solution to this issue—like for CPUs—would be to add dynamic scheduling of fragments to GPUs to detect and avoid pipeline hazards.

The Unified Render Architecture proposed by ATI in the next generation GPU architecture [18] aims for better load balancing on vertex and fragment shaders by considering them as a single shader unit that is managed by a thread arbiter (or the fragment dispatcher) . The thread arbiter controls the data being passed to the shader units, and finds the best possible way to ensure that all of the shader units are busy. If the thread arbiter can be configured, or even programmed in the future, it would be possible to divide fragments into nonoverlapping groups that are processed by different shader units. Similarly to early-*Z* culling [45], the hardware could keep a scoreboard [69] in the form of a coarse image of what fragments are currently in the pipeline. For a given incoming fragment packet (fragments are currently packed to perform derivative operations), the scoreboard would be checked for overlaps with a fragment already in the pipeline. If an overlap is detected, the packet could be inserted in a buffer, and the next packet coming from the rasterizer could be tested. An overflow of this temporary buffer would result in a stall until a pipeline becomes available.

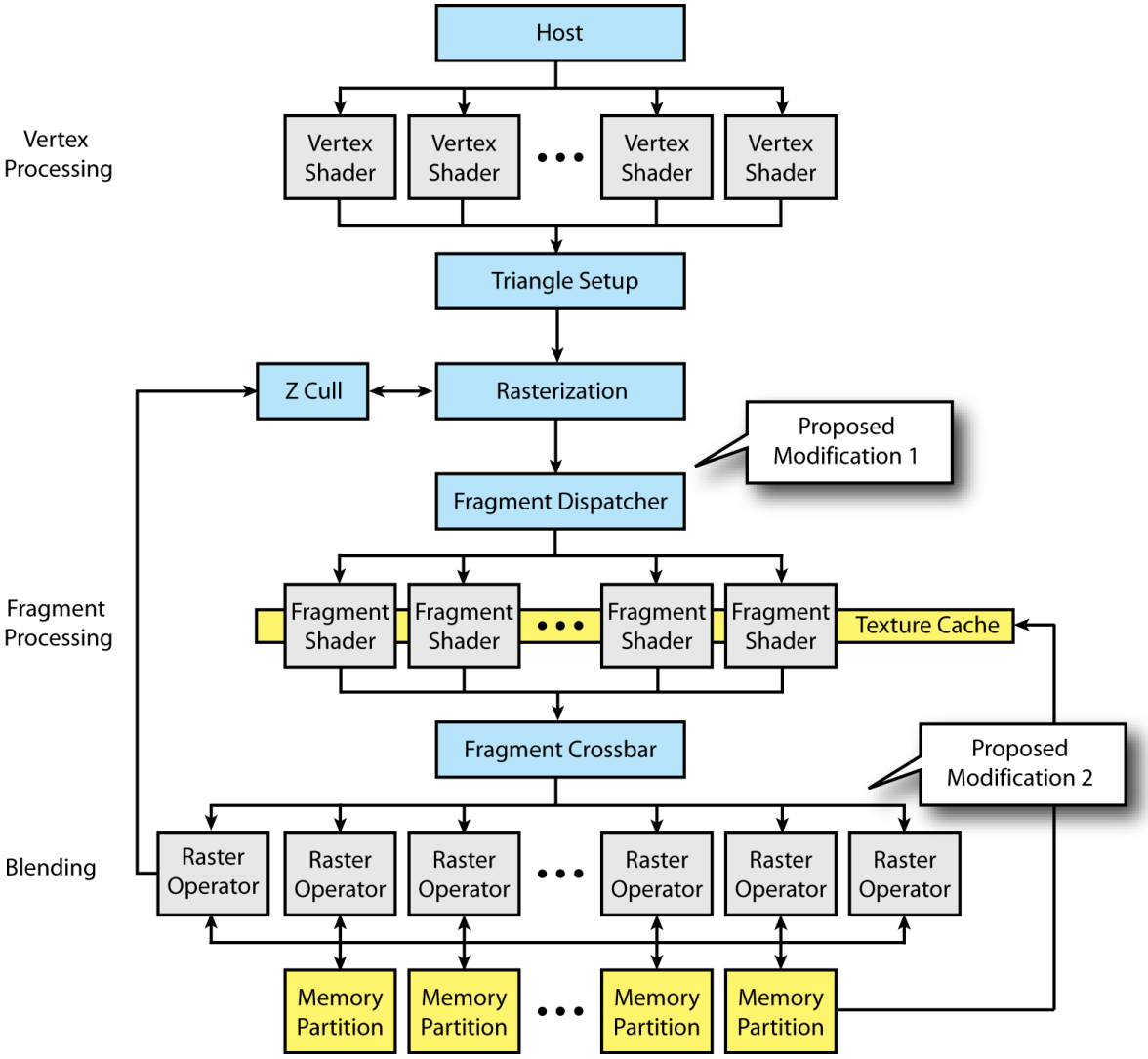


Figure 4.1. The GPU pipeline of the GeForce 6/7 showing where our proposed modifications will occur. Figure adapted from [45].

The main advantage to *k*-buffer support at the rasterization stage is that it leaves the current pipeline relatively unchanged. Another advantage is that *k*-buffer programs can use all the features of fragment shaders, including texture accesses (e.g., for lookup tables). Note that the *k*-buffer access does not need to be a texture access. It may be more efficient to implement it as varying arguments like texture coordinates. In any case, *k*-buffer programs could mix texture accesses with *k*-buffer accesses. However, there are several disadvantages to this proposed solution. First, by modifying depth in fragment programs, early-*Z* tests are invalidated. Since the *k*-buffer generally requires all the fragments anyway, this is not a major issue. Second, the fragment scheduler may adversely affect performance by reducing the parallelism during fragment processing. Finally, full-screen antialiasing presents some

challenges because fragment shaders operate on pixel fragments while multisample antialiased blending operations occur on multiple samples per pixel [10]. To support antialiasing with the k-buffer would likely require supporting the more costly supersample antialiasing rather than the more efficient multisample antialiasing.

A more promising approach is to implement the k-buffer by allowing programmable blending. Currently, the only RMW operations on colors in the graphics pipeline are fixed-function per-pixel operations. The k-buffer can be supported by extending RMW capabilities using blending programs similar to fragment or vertex programs. Specifying different k-buffer applications could then occur with the use of a programmable blender that takes as input the results of the fragment shader and outputs the values in the k-buffer for the pixel. Programmable blending has been discussed as a possible hardware extension in the future [10].

To demonstrate that this model conceptually fits into the current pipeline, we extended Mesa 6.5 with k-buffer support for programmable blending. We modified the OSMesa driver, which is a purely software implementation of Mesa. The following changes to the existing pipeline were made. In the software rasterizer, we added a k-buffer mode that changes the behavior of MRTs. When in this mode, fragments leaving the fragment program are passed to a programmable blender. A programmable blender is a specialized fragment program which takes as input the current pixel in the framebuffer, and the current k-buffer fragments for this pixel, passing these fragments as four-float varying arguments.

Implementing k-buffers with programmable blending has a number of benefits over implementing them in the fragment program stage. First, it does not require texture (random) memory access, which means the only memory reads are pure stream accesses from the incoming fragment and the k-buffer. Second, it does not require scheduling so it would not affect the parallelization of the fragment pipeline. Third, current caching strategies for pixel tiles are critical to GPU performance [31] and would still be applicable. Fourth, no cache coherency would be required between the pixel tile caches and the texture caches. Finally, multisample antialiasing would still be possible, given that the k-buffer operates directly on subsamples rather than fragments. One consideration of this approach is that many compression algorithms are hardcoded for the fixed semantics of each component of the framebuffer [31]. By generalizing the framebuffer, the hardware may no longer know which chunks of memory can be optimized for depth, stencil, color, etc.

4.3.2 Current Hardware Implementation

We created an experimental implementation of the k-buffer using current hardware to test k-buffer applications and demonstrate the flexibility of the framework. All of the effects and results shown in this chapter were created using this implementation, unless specified otherwise. Our experimental k-buffer is implemented in OpenGL as a set of textures that can be read and written to in fragment programs using MRTs, as described in Section 4.3.1. Since current hardware does not handle RMW pipeline hazards, artifacts may appear. To perform off-screen rendering into the MRTs with OpenGL, we use a Framebuffer Object (FBO), which is a collection of logical buffers such as color, depth, or stencil. Currently, up to four color buffers can be attached to an FBO and used as MRTs. Algorithms operating on the k-buffer are currently implemented as fragment programs on FP32 textures with Z-culling disabled.

If the desired application requires streaming with programmable blending, one of the color attachments may act as an off-screen framebuffer, while the other three contain k-buffer entries. Otherwise, all four available color attachments may be used to store the k-buffer entries. These entries can be single values (e.g., depth), or sets of values (e.g., depth, scalar, color, etc.) depending on the application. Thus, the number of values per entry directly effects the size of k available for the application. With four MRTs, it is possible to store up to 16 fragment attributes using RGBA textures. The precision of these color attachments is application specific, thus by quantizing the values, it is possible to pack additional k-buffer entries into the color attachments. To minimize the number of attributes stored with each k-entry, we would ideally need to store only one depth value per entry. The world-space position can be reconstructed from the depth (either the clip-space depth or the distance to the eye). From the positions, normals can be estimated using central differencing so deferred shading can be performed. Using lookup table IDs can also be useful to reduce the number of attributes stored in the k-buffer.

Our experimental k-buffer reads and writes from the same textures in each fragment operation. This is available in the current OpenGL API even though the results are undefined. In practice, this may result in RMW hazards due to the parallel nature of GPU architectures (for example, see Figure 4.2). To reduce these hazards, we have developed two heuristics applied to scene geometry prior to rasterization to avoid screen-space overlaps. Depending on the application and scene, these heuristics may be necessary for correct images using current hardware. Our first heuristic is to sort the primitives by their centroid depth. This effectively layers the geometry in screen space (see Figure 4.3), which reduces the likelihood

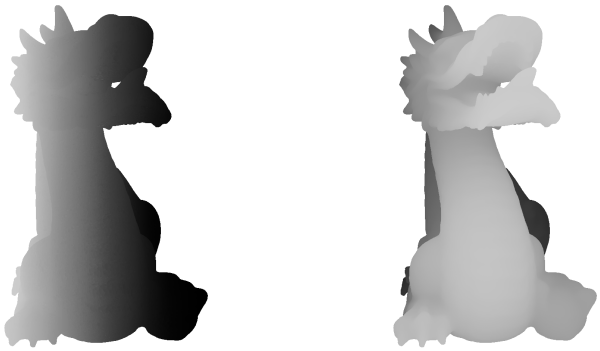
of overlapping fragments in the pipeline simultaneously. For algorithms that require complete sorting of the fragments, this object-space sorting is already required, thus the hazards are inherently reduced without additional penalty. Our second heuristic is to batch triangles and flush the graphics pipeline after each batch by rendering a full-screen quadrilateral with a `GL_FALSE` color mask. For performance reasons, we use a simple algorithm to create the batches—triangles are added to the current batch in order, until a maximum batch size is reached. Though these heuristics adversely affect performance by reducing the caching on the GPU and stalling the GPU pipelines, they demonstrate the ability to overcome the RMW hazards that occur with the current hardware implementation.

4.4 Single-Pass Depth Peeling

Depth peeling captures multiple depth layers by stripping the visible layers of fragments in multiple peeling passes using Z-buffer tests. The k-buffer makes it possible to perform up to k Z-buffer tests in a single geometry pass and therefore capture the first k fragments along a viewing ray. Effects that use depth peeling include transparency, translucency, constructive solid geometry, midpoint shadow mapping, and volume rendering. A k-buffer can be used to perform single-pass depth peeling by storing depth-sorted fragments (see Figure 4.4). The depth values of the k-buffer entries are initialized with the largest possible depth value. Upon rasterization, each fragment is inserted into the k-buffer in increasing depth order. Thus, the fragments are captured in depth order using an insertion sort, which is an efficient algorithm for small k sizes. Another appropriate algorithm would be a bubble sort.

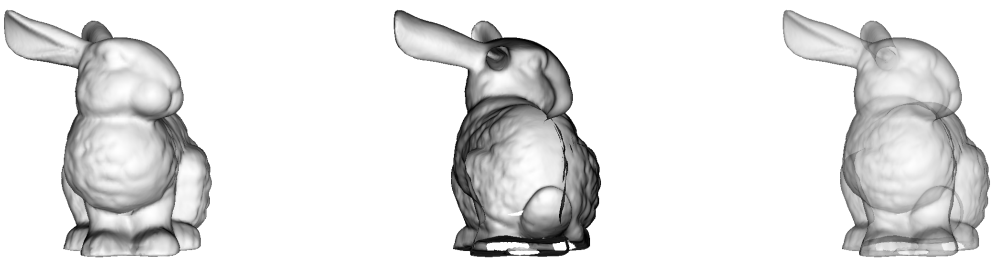
(a) (b) (c)

Figure 4.2. Artifacts that appear in our current implementation due to hazards. (a) With the original mesh order. (b) With the depth sort. (c) Difference image.



(a) (b)

Figure 4.3. Visualization of the rendering order. The triangles are colored in order from gray to black. (a) With the default mesh order. (b) Ordering the triangles before rendering with a depth sort by centroid.



(a) (b) (c)

Figure 4.4. Depth peeling two layers from the dragon dataset. (a) First layer, (b) Second layer, (c) Transparency using four layers.

CHAPTER 5

ROBUST SOFT SHADOW MAPPING

5.1 Introduction

Invented in 1978 by Williams [75], shadow mapping is an image-based algorithm for rendering hard shadows that maps very well to GPUs. In this classic algorithm, a preliminary pass renders the scene from the viewpoint of the light into a depth buffer (dubbed a *shadow map*). Then, image fragments are transformed to the image space of the light and their depths are compared to the shadow map depths to determine if the fragment is occluded from the light. The main advantages of shadow mapping are its speed and simplicity. In 1990, the accumulation buffer by Haeberli and Akeley [29] enabled hardware-accelerated soft shadows by averaging multiple hard shadows. This was implemented using shadow mapping by Heckbert and Herf [32] and can be performed efficiently on current GPUs, using additive blending. Still, it is costly for complex scenes because it requires in the order of a hundred passes over the scene geometry to produce smooth soft shadows. Therefore, recent research efforts in soft shadow mapping have focused on rendering soft shadows with a minimum number of geometry passes. Here, we cover only the work the most related to our approach.

5.1.1 Single-Layer Approaches

Various efforts have been made on approximating soft shadows from a single shadow map (for a recent survey, see [30]). All of the approaches based on a single shadow map have light bleeding issues in overlapping shadows from different occluders because of gaps between the shadow map pixels that are seen from some other regions of the area light.

Atty et al. [2] recently proposed a soft shadow mapping algorithm based on backprojection, which renders interactive physically-based shadows on the GPU by separating shadow casters and shadow receivers into two shadow maps. For each occluder, it finds all the associated receivers, sums up the occlusion ratio of the occluder over its receivers, and stores the result in a soft shadow map which is projected onto the scene. This algorithm does not support self-shadowing. An improved soft shadow mapping algorithm with self-shadowing was re-

cently proposed by Guennebaud et al. [27]. It addresses many of the issues of previous approaches, including light bleeding. However, its gap filling strategy may remove some important details in penumbra. A similar algorithm was independently proposed by Aszdi and Szirmay-Kalos [3], which supports self-shadowing but does not address light bleeding.

5.1.2 Multiple-Layer Approaches

The problem of soft shadow rendering is to determine what portion of the light is occluded from a given shading point. The input to this problem should be a representation of the scene that produces a final result as close as possible to the full geometry. Some prevalent algorithms for soft shadows use multiple layers of shadow maps as a representation of the scene. Keating and Max [43] capture multiple depths per layers in a buffer that they refer to as a multilayer depth image, based on earlier work on image-based rendering from Max and Ohsaki [54]. This data structure is a type of Layered Depth Image (LDI) [65]. They noticed that gaps in image-based 3D model reconstructions were reduced by capturing multiple layers per view, instead of just the nearest layer, and used the additional layers for removing light bleeding in their soft shadow mapping algorithm [43]. Since the light samples are close together and the LDI only stores points from opaque objects directly visible from the light, the average number of required layers is much less than for an LDI that represents the entire scene. In fact, for most complex scenes, four layers are sufficient [1].

A variation of this approach by Agrawala et al. [1] warps shadow maps taken from multiple points on the light surface into a layered attenuation map, which is a type of LDI. Each layer of a layered attenuation map stores a depth and the percentage of the light that is visible from a point. Im et al. [39] build an LDI from a single view point at the center of the light and use image warping to compute occlusion values stored in the LDI. The main disadvantage of these two warping approaches is that they require reading and writing to different pixel coordinates, which cannot be done with current graphics hardware. This makes it difficult to achieve real-time performance in dynamic scenes. Recently, Eisemann and Décoret [21] showed that plausible soft shadows can be rendered in real-time by approximating an object by 4 to 16 slices based on a regular partition of depth. They achieve very fast real-time performance by prefiltering occlusion maps similarly to mipmapping. As for Keating and Max [43], the depths of the fragments rasterized from the light are quantized by clamping, which introduces error. However, this error may be acceptable if performance is more important than quality.

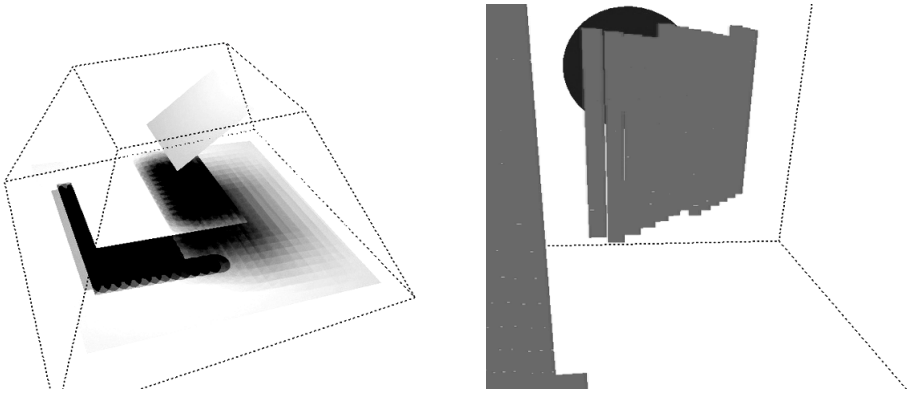
Our algorithm is most similar to the approach of Im et al. [39]. We represent the visibility of an area light from a single point at the center of the light, using multiple depth layers, similar to Keating and Max [43], Agrawala et al. [1], and Im et al. [39]. Like Im et al., we extract our layers using depth peeling [22], which can be performed efficiently on graphics hardware. This allows us to remove most light bleeding artifacts without extending the shadow map samples. Therefore, our algorithm has less overshadowing issues than the one of Guennebaud et al. [27]. The technique of Im et al. handles light bleeding without overshadowing, but requires parameter tuning and does not map completely on current GPUs. We propose two ways to handle self-shadowing robustly. The first way is an extension of midpoint shadow mapping [79] to multiple layers, as described in our technical report [6], which has a minimal number of intrinsic parameters. However, minor artifacts remain at grazing angles and the method fails for thin objects. To add more robustness to our algorithm, we combine midpoint shadow mapping with depth gradients [62]. This makes it possible to remove all surface acne in all cases at the expense of adding two parameters to the algorithm. The result is a more automatic algorithm for generating physically-based soft shadows using shadow mapping.

5.2 Soft Shadow Mapping Artifacts

Due to the discrete nature of shadow maps, small overlaps and gaps may occur between shadow map pixels seen from a given shading point (the point on the surface being shaded). Gaps result in light bleeding, whereas overlaps result in overshadowing. In addition, surface acne appears because the points seen from the eye do not correspond exactly to the points seen from the light. We do not address the problem of removing occlusion overlaps. However, we do address the issues of light bleeding and surface acne.

5.2.1 Light Bleeding

Light bleeding occurs when there is a gap between shadow map pixels seen from the shading point. See Figure 5.1. Thus, surfaces that are in the penumbra of the first object, as seen from the light, will not get shadows from other objects that may fully occlude the surface. This is a significant problem for any scene with multiple overlapping shadows and is magnified with higher depth ranges. Figure 5.1 shows an example of this case where objects closest to the light interfere with the shadows cast from objects closer to the shadow. One approach to resolving the problem of light bleeding is to overestimate the shadow by extending the shadow



(a) (b)

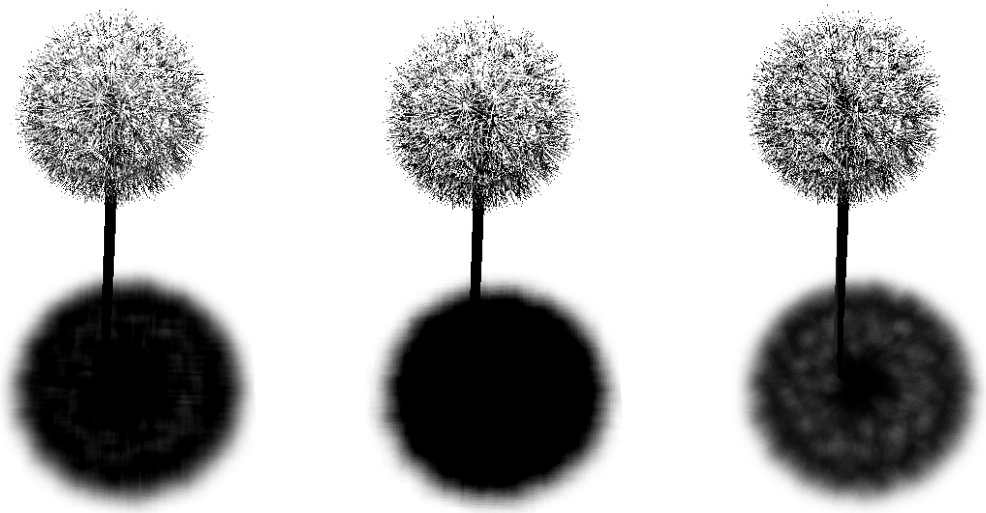
Figure 5.1. Visualization of the gaps between shadow map samples. (a) General view of the scene showing the shadow frustum in dashed lines, and the result of a soft shadow algorithm using a traditional single-layer shadow map. (b) Visualization of the shadow map fragments unprojected into the world, looking at the light from an area that should be completely in shadow.

map pixels [27]. This *gap filling* approach does not work properly for shadows of thin objects such as hair (see Figure 5.2).

Our solution to light bleeding is to search for occluding samples in all the layers from the deepest shadow map layer to the first shadow map layer (similar to shadow ray traversal in ray tracing). This has the effect of removing the discontinuities in the sampling of the visibility of the light which minimizes light bleeding artifacts. Our solution is described in more detail in Section 5.3.1.

5.2.2 Surface Acne

Self-shadowing is traditionally handled by computing a depth for the shading point (z) and for the shadow map sample (z_s). These depths are computed in world-space by measuring the distance from the light plane to the samples. A shading point is considered in shadow if and only if $z_s < z$. However, due to the depth quantization and the discrete nature of the shadow map, *surface acne* (false self-shadowing) appears in the final image. To counter this, a depth bias is often applied to move the occluder farther away from the shading point. The shadow test then becomes $z_s + bias < z$. A depth bias that is too small results in surface acne, while a depth bias that is too large results in incorrectly placed shadows. Unfortunately, the depth bias often requires manual tuning to achieve good image quality. Figure 5.3 shows the errors that occur when using a uniform depth bias to correct surface acne.



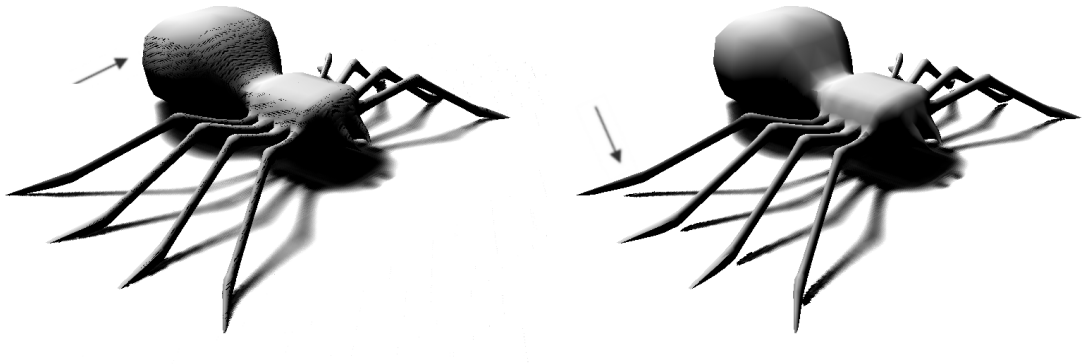
(a) (b) (c)

Figure 5.2. Gap filling. The dandelion scene shows that gap filling (Guennebaud et al. [27]) can result in shadows that are too dark. (a) Backprojection without gap filling. (b) With gap filling. (c) A ray-traced image of the dandelion scene shows the correct shadowing.

Several approaches have been developed to adaptively compute the depth bias. One common method is to use the `glPolygonOffset` function in OpenGL. This function offsets the depth of the rasterized fragments by a constant epsilon and a coefficient proportional to the maximum slope of the depth values around the pixel. This method requires tuning the slope parameter to remove surface acne. With the Shader Model 3.0, the slope of a custom depth value can now be computed in a fragment shader.

The midpoint shadow map algorithm [79] uses the average depth of the first two surfaces encountered from the viewpoint of the light for the self-shadowing test. This algorithm works for both closed and nonclosed objects. Using a midpoint shadow map removes most of the issues with self-shadowing. However, it has two minor issues. First, light bleeding may appear when the midpoint and the original point are too far away. This can be fixed using a maximum depth bias [73]. Second, surface acne may remain at corners where the midpoint and the original surface meet. We found that these artifacts are minor in relation to the artifacts that are inherent to a limited-resolution shadow map; thus we use an extension of midpoint shadow mapping to handle self-shadowing. Our solution is described in more detail in Section 5.3.4.

A different way of handling self-shadowing without depth comparison is the priority buffer [35]. In this technique, unique ids are assigned to objects in the scene and the id of the shading point is compared with the id of the shadow map sample to determine self-shadowing. The approach adds more complexity to the application since every primitive must have an



(a) (b)

Figure 5.3. Self-shadowing issues, using a uniform depth test. (a) A depth bias too small results in surface acne (bias = 0.01). (b) A depth bias too large results in the incorrect placement of shadows (bias = 0.3).

object id associated with it. A general method is to use an object id per triangle. However, in this case, surface acne appears at triangle boundaries where the shading point and the sample are on different triangles. Fernando proposed using depth comparisons in the cases where the triangle ids fail [23].

5.3 Algorithm

We assume a square light source and square shadow map pixels. Our algorithm proceeds as traditional shadow mapping by rendering the scene from the light to create the shadow map, then from the viewpoint. To create a layered shadow map, for every frame we render the entire scene multiple times from the light using depth peeling [22] with a fixed number of layers (see Figure 5.4). Each layer contains the world-space distances to the light plane. After the shadow map passes, the shadow intensity of each pixel in the image is computed by projecting the pixel onto the near plane of the shadow map and testing shadow map samples for occlusion. A conservative search region is computed like in [27]. For each sample coordinate in the search region, the corresponding samples are backprojected onto the light, and the contribution of the nearest occluding sample to the shading point is selected. Each aspect of the algorithm is described in greater detail in this section.

5.3.1 Layered Shadow Map

Each sample coordinate in the shadow map corresponds to a beam with the origin at the center of the light, going through the shadow map sample (see Figure 5.4). By sampling the

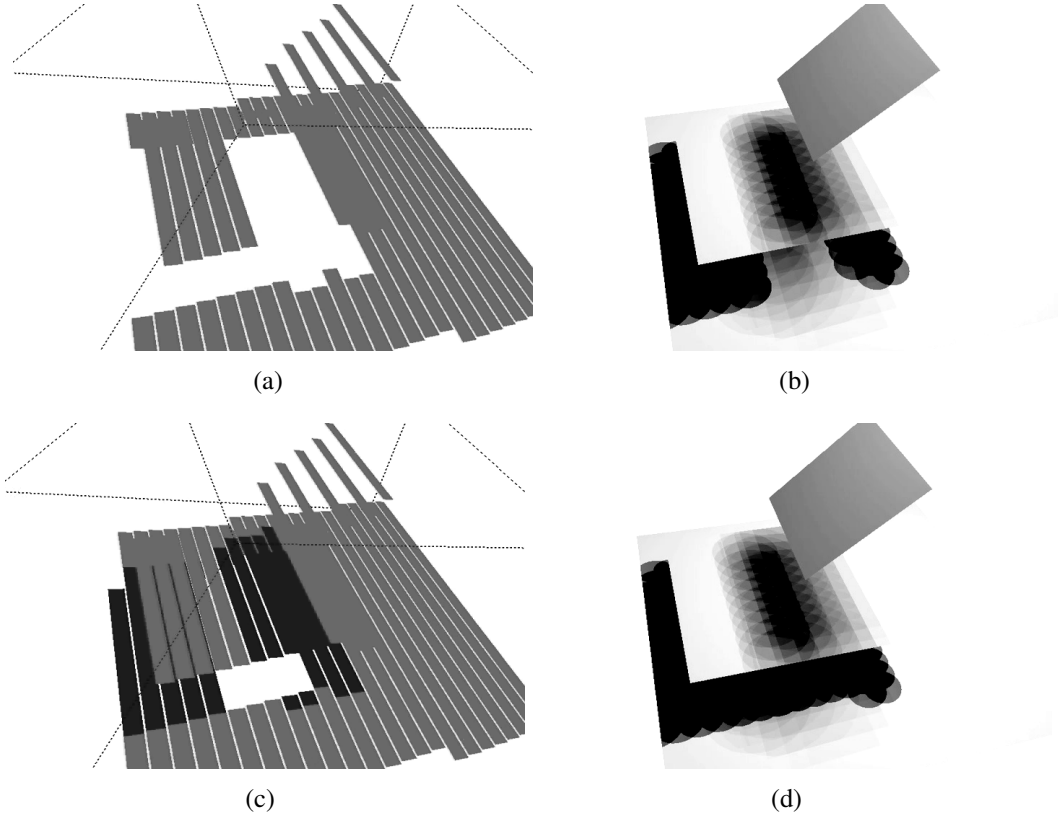


Figure 5.4. Reducing light bleeding with a layered shadow map. First row: with one shadow map layer. Second row: with two layers. Left column: shadow map visualization. Right column: soft shadows.

depth values of the first k samples for a given shadow map sample coordinate, we effectively take the first k hits along the corresponding light beam. We efficiently reduce light bleeding by computing the occlusion of every sample along a light beam and using the furthest occluding sample from the light. In practice, we found that by using only three layers we can remove most light bleeding artifacts, as was noted by Agrawala et al. [1]. For a given sample coordinate in the shadow map, we sample a fixed number of depths in our layered shadow map. We then compute biased depth values corresponding to the midpoints between the layers. We use the biased depth values for handling self-shadowing and the original depth values for computing the actual shadow contributions. Then, the samples are processed starting from the deepest shadow map layer and the first nonzero shadow contribution is selected.

5.3.2 Search Region

To compute a conservative search region, we use the same iterative algorithm as Guennebaud et al. [27]. The initial search region is the intersection of the light plane with the shadow frustum from the shading point to the square light. The center of the search region

is the projection of the shading point onto the near plane of the light frustum. The width (in pixels) of the region is $kr(n)$ with:

$$kr(z_{min}) = r\ l\ \frac{n}{w}\ (\frac{1}{z_{min}} - \frac{1}{z}) \tag{5.1}$$

Next, the we find the local minimum depth in the current search region using a hierarchical shadow map (HSM) built over the first shadow map layer. The HSM is sampled at the level $\lceil log_2(kr) \rceil + 1$, where 0 is the full resolution level, 1 is the first hierarchy level with half the resolution, and so on. The minimum depth z_{min} over the four nearest neighbors at this level of the HSM is computed, and the search region width is reduced to $kr(z_{min})$. This reduction step can be applied multiple times. In practice, we found that two iterations was a good quality/performance tradeoff.

5.3.3 Adaptive Sampling

Apart from the shadow map resolution, the main quality parameter of our algorithm is a maximum number of samples per pixel *max nspp* (see Figure 5.5). We compute the width of the square search region as described in Section 5.3.2. If the number of samples in the search region is less than *max nspp*, we process all the sample coordinates in the search region. Otherwise, we use an adaptive step so that the search area is sampled uniformly with max_{nspp} samples. We compute $step = (region\ nspp)/(max\ nspp)$, and we scale the world-space size of the samples w_s by $step$ when backprojecting the samples. This has the effect of multiplying the stride of the shadow map by $step$, and dividing kr by $step$.

(a) (b) (c)

Figure 5.5. Adaptive sampling, on the thin tree scene rendered using our algorithm from a three-layer 1024^2 shadow map. Image resolution: 800x800. (a) $max\ nspp = 289$, 4.1 fps. (b) $max\ nspp = 1089$, 1.2 fps. (c) ray tracing with 1,000 shadow rays per pixel.

5.3.4 Adaptive Depth Bias

In this section, we present our hybrid technique handling self-shadowing with an adaptive depth bias, based on a combination of two state-of-the-art techniques: midpoint shadow maps [79] and depth gradients [62].

Since the algorithm is using multiple depths per shadow map pixel to help removing light bleeding, we can use these depth values to compute biased depth values. The idea of midpoint shadow mapping [79] and second-depth shadow mapping [71] is that for each shadow map pixel, any depth between the first and second depths can be used instead of the original first depth. Midpoint shadow mapping takes the middle of the first and second, while second-depth shadow mapping takes the second depth. In practice, using the second value is more robust against surface acne, but introduces more light bleeding. Therefore, we use midpoints between consecutive depth values.

Unfortunately, there are cases where the midpoints are insufficient, for example a scene with two thin slabs such as the windshield of a car. Therefore, we developed a version of the depth gradient technique presented by Schuler in ShaderX 4 [62]. This approach is applicable to any rasterizable geometry, including points and lines. For each shadow map pixel, we compute the horizontal and vertical gradients $ddx(z)$ and $ddy(z)$, as the depth differences between z and a nearest neighbor in shadow map space, horizontally or vertically respectively, as computed by the `glPolygonOffset` OpenGL function implemented in hardware. The scalar depth gradient $dz(z)$ is:

$$dz(z) = max(|ddx(z)|, |ddy(z)|) \tag{5.2}$$

This gradient is then scaled by a constant *gscale* and optionally clamped by a maximum value *gmax*. The *gscale* parameter is similar to the first parameter of the `glPolygonOffset` function. The *gscale* parameter typically needs to be between 1 and 10 to remove all surface acne. High values remove all surface acne but may introduce light bleeding, because valid occluders are ignored. The *gmax* parameter, as suggested by Schuler [62], is useful for avoiding light bleeding. Since the gradient is in world units, the *gmax* parameter can be set according to the scale of the objects in the scene. To make the algorithm more robust for low values of *gscale*, we use an hybrid bias equal to the maximum of the midpoint depth bias and the slope-based depth bias.

5.3.5 Backprojection

Each shadow map sample is first tested for approximate occlusion using a traditional depth test on its biased depth value. This test is necessary to handle self-shadowing and to not take into account samples behind the shadowing point. If the sample passes the test, we backproject it onto the light based on its actual depth value, using the backprojection algorithm of Guennebaud et al. [27], but without gap filling. For completeness, we sum up this algorithm. The normalized coordinates B of the backprojection bounds of a sample of depth z_s seen from a shading point of depth z is:

$$B = \begin{bmatrix} b_{left} \\ b_{right} \\ b_{bottom} \\ b_{top} \end{bmatrix} = \begin{matrix} (du-0.5) \\ (du+0.5) \\ (dv-0.5) \\ (dv+0.5) \end{matrix} (\frac{w}{n\,r}\,z_s)(\frac{z}{z-z_s})\frac{1}{l} \tag{5.3}$$

where (u_s, v_s) and (u, v) are respectively the shadow map coordinates of the sample and the shading point (in pixels), $du = u_s - u$, $dv = v_s - v$, $w_s = \frac{w}{n\,r}\,z_s$ is the size of the sample (in world-space), w and n are respectively the width and depth of the light plane (in world-space), r is the width of the shadow map (in pixels), z is the depth of the shading point (in world-space), and l is the width of the light (in world-space). The intersection of the backprojected sample with the light is performed by clamping B by [-0.5,0.5]. The shadow contribution of a sample is the area of the clamped backprojection area $A = (b_{right} - b_{left})(b_{top} - b_{bottom})$.

5.4 Discussion

For a given viewpoint, the speed of the algorithm depends on the number of nonempty and front-facing pixels (nonblack color), the resolution of the shadow map, the maximum number of samples per pixel, the number of depth layers, and the number of iterations of the search region reduction algorithm (in our examples, we use two iterations). Increasing the number of shadow map layers darkens the shadows and removes light bleeding. For all our scenes, we used three shadow map layers and achieved images that look similar to ray-traced images (see Figure 5.6). We believe that this would be useful for previsualizating soft shadows in computer graphics films. Though the algorithm we have introduced removes most light bleeding and surface acne artifacts that we describe in Section 5.2, some issues may still arise.

5.4.1 Gaps and overlaps

Our solution to light bleeding is not guaranteed to remove all light bleeding since it uses a fixed number of layers. Furthermore, for a simple scene such as a quad nearly tangent to the

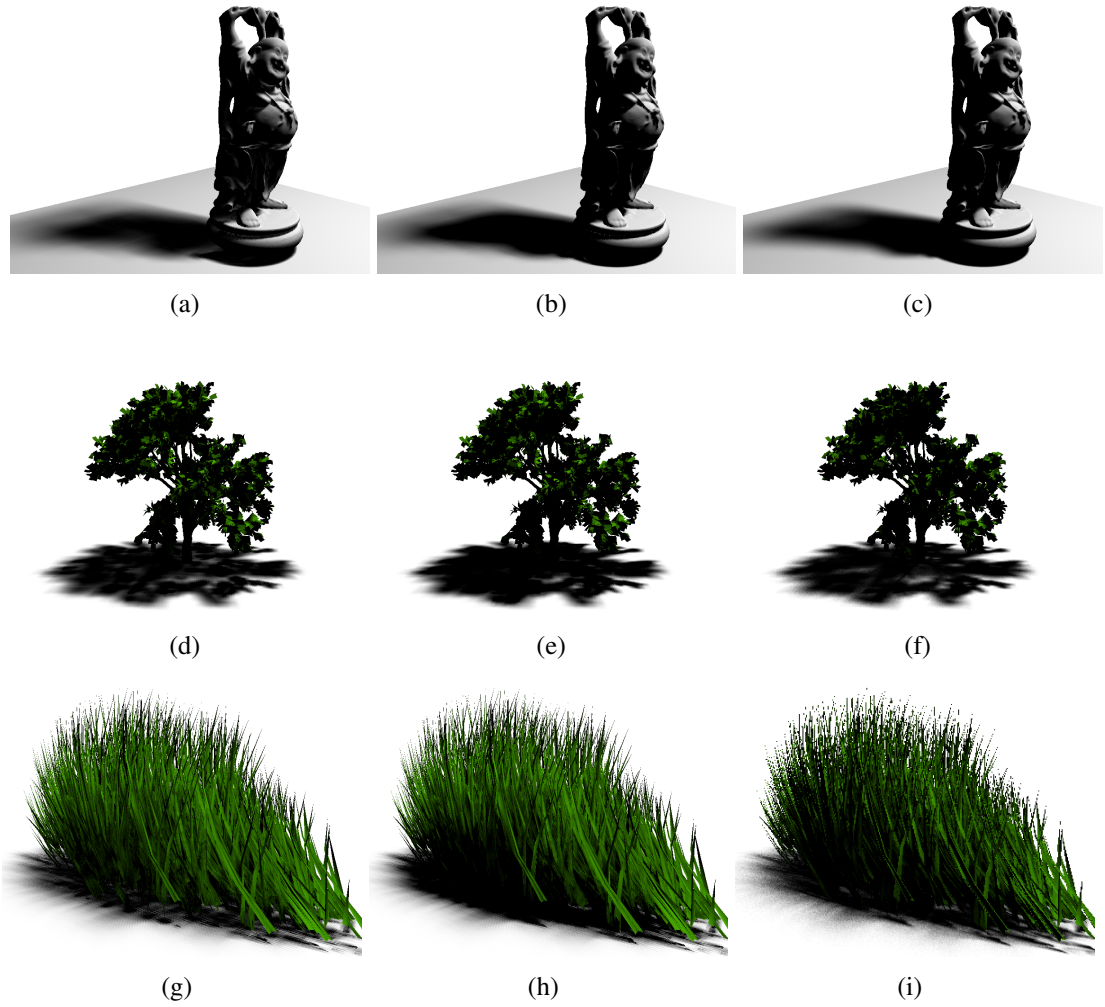
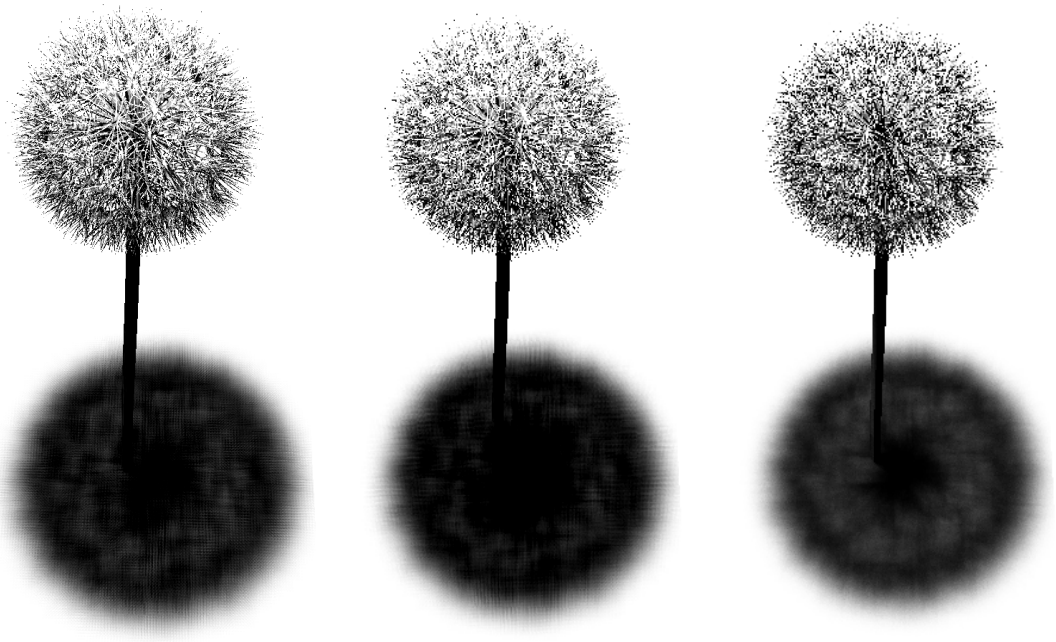


Figure 5.6. Adding shadow map layers. Our algorithm with uniform sampling ($max\ nspp = 441$) compared to ray tracing with 1,000 samples per pixel, using both the midpoint depth bias and a slope-based bias. Image Resolution: $800x600$. Shadow Map Resolution: 1024^2. GPU: GeForce 7800 GTX, CPU: AMD Opteron Processor 275 @ 2.2 Ghz. **Left:** Our algorithm with one layer. **Middle:** Our algorithm with three layers. **Right:** Ray tracing. **First row:** Happy Buddha (293,264 triangles). **Second row:** Thick Tree (27,869 triangles). **Third row:** Weed (26,195 triangles).

light rays, light may still bleed though. Inversely, because the backprojections of shadow map samples may overlap, the algorithm tends to overshadow (see Figures 5.5 and 5.7). As the resolution of the shadow map or the number of samples increases, this problem becomes more pronounced. Thus, our solution improves on the gap filling solution of Guennebaud et al. [27], but still is not guaranteed to remove all light bleeding and still overshadows. With multicore CPUs such as the Cell processor, and with unified memory architectures such as the XBox 360 and the PS3, hybrid CPU-GPU techniques based on image warping [39, 1] which address the problem without overshadowing have become more practical for interactive rendering.

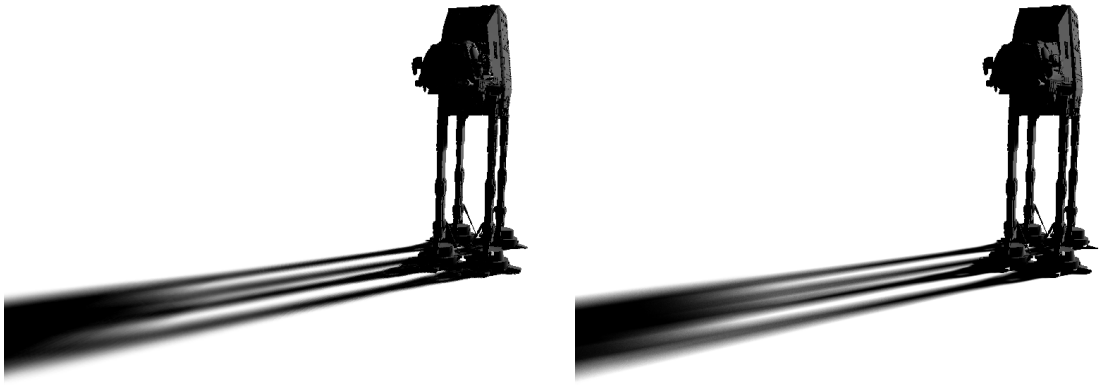


(a) (b) (c)

Figure 5.7. Dandelion (35,107 triangles). Using three layers. Overshadowing because of overlaps between fragments. Image Resolution: 512*x*512. **Left:** Our algorithm with 169 max spp. **Middle:** Our algorithm with 441 max spp. **Right:** Ray tracing with 1,000 spp.

5.4.2 Surface acne

As described in Section 5.3.4, although the algorithm is using midpoint shadow mapping, surface acne may appear if the *gscale* parameter is too small, and light bleeding may appear if it is too large. Midpoints fail to remove surface acne when the shadow map pixels are bigger than the space between two consecutive depth layers. In this case, shadow map pixels can still peak through the surface [62], generating surface acne. The ideal solution would be to increase the resolution of the shadow map such that the pixels become smaller than the space between layers. However, this may not be practical. The issue can be handled independently of the shadow map resolution by using the *gmax* parameter, which clamps the depth bias. We believe that it is better for our algorithm to have a few parameters which can be set to default values for most scenes ($gscale = 0$, $gmax = infinity$), rather than not being able to handle occasional surface acne. Figure 5.9 shows that a slope-based bias helps remove surface acne in cases where the midpoints fail.



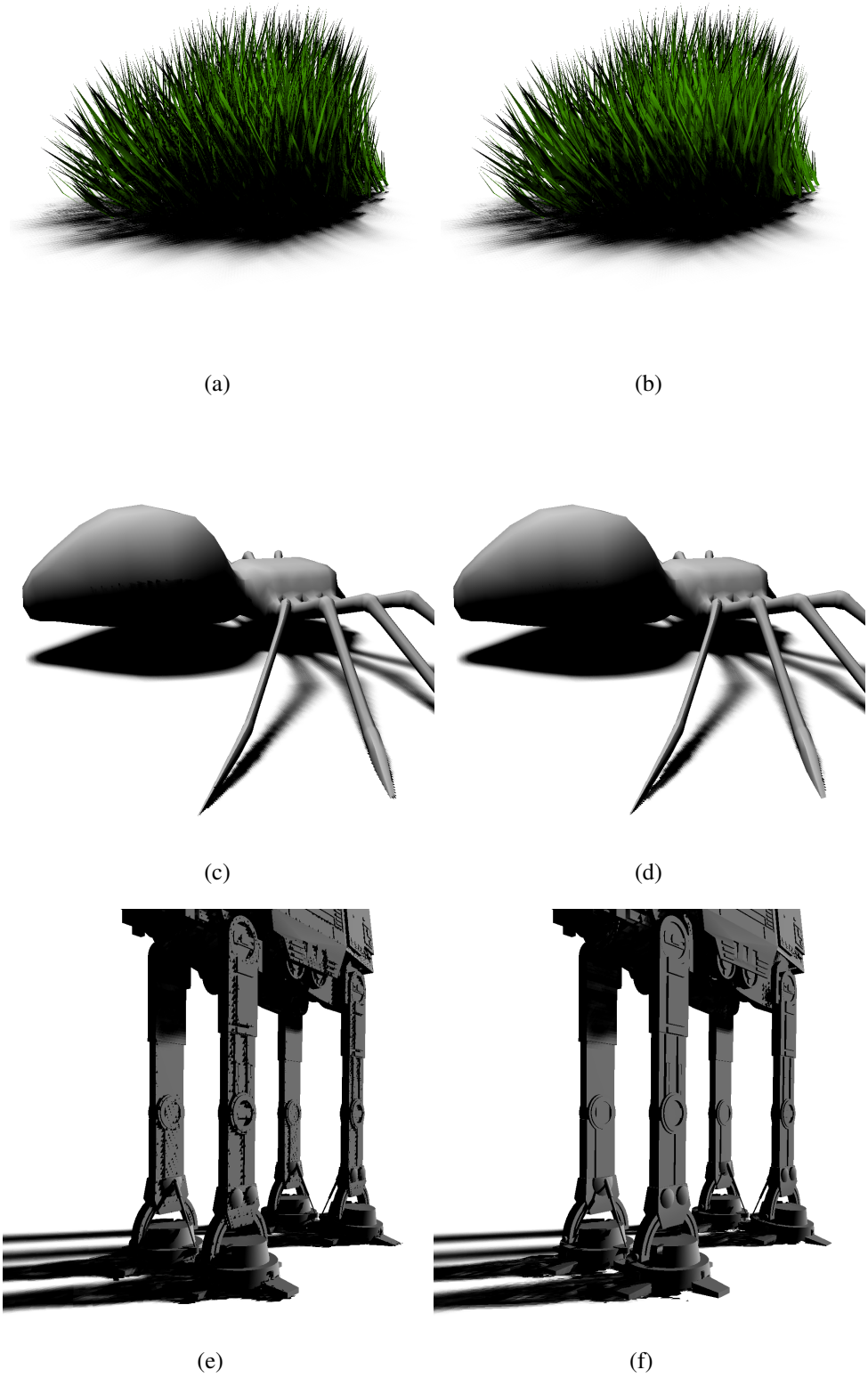
(a) (b)

Figure 5.8. AT-AT Walker (211,140 triangles). Image Resolution: 800x600. (a) Using our algorithm with 3 shadow map layers and midpoint biases only, 441 max spp. 5.9 fps. (b) Ray tracing. 17 min. GPU: GeForce 7800 GTX, CPU: AMD Opteron Processor 275 @ 2.2 Ghz.

5.4.3 Undersampling

Because of the sampling nature of the algorithm, other quality issues may arise. First, the resolution of the shadow map may be too low in which case aliasing appears when zooming in. Undersampling in the shadow map results in jaggies as shown on Figure 5.9. This problem is a well known problem with shadow mapping. The effective resolution of a shadow map can be increased by warping the view of the shadow map [76, 53], or by using multiple shadow maps for the same light frustum [24, 49, 46]. Irregular shadow maps [16, 40] eliminate the problem of aliasing, but cannot be implemented on current GPUs because they require an irregular rasterizer. Another common technique is to blur the shadow map. Blurring our shadow map would help remove aliasing, but may introduce light bleeding artifacts, as it is the case with variance shadow maps [19].

Second, our uniform sampling scheme may introduce banding artifacts because the samples are too coherent. If this becomes noticeable, a solution is to precompute a texture with jittered or Poisson offsets in a unit disk and to scale the offsets by the radius of the search area to sample the layered shadow map. Poisson sampling can be combined with dithering, to further hide artifacts. Another issue may arise due to insufficient detail in the hierarchical shadow map (HSM) optimization of the search region algorithm, as noted by Guennebaud et al. [27]. Because the search area optimization is based on a hierarchical shadow map, the size of the search area may jump abruptly from one region to the next, which can introduce discontinuities in the sampling. In this case, disabling the search area reduction may improve the image quality, trading discontinuities for banding.



(a) (b)

(c) (d)

(e) (f)

Figure 5.9. Self-shadowing artifacts. **Left column:** With midpoint-based bias only. **Second column:** Our hybrid technique using the maximum of the midpoint-based bias and a slope-based bias.

5.5 Implementation Details

5.5.1 Data Flow

Since the main bottleneck in the algorithm is computing per-pixel shadows, we use a deferred shadowing pipeline which computes the shadows only for the visible fragments. To do this, we render the scene from the eye and store the world-space positions and shaded colors of the pixels in two 16-bit RGBA floating-point textures using a framebuffer object with two render targets. For the position map, the x, y, and z world-space coordinates of the samples are interpolated from the vertex positions and stored in the R, G, and B channels. The A channel is used to differentiate background pixels. In addition to the deferred shading, we use a predicate in the shadow shader to avoid computing shadows of background pixels or pixels with black diffuse colors.

We use one 16-bit RGBA floating-point texture per layer to store the layered shadow map. On GeForce 6 and GeForce 7, this is the only renderable texture format with 16-bit floating-point precision per channel. Using a 16-bit format is important to reduce bandwidth. When rendering the scene from the light, the R channel is used to store the depth of the samples. We then do an additional pass to pack four depths into a single 16-bit RGBA floating-point texture. This allows us to fetch four depth values with a single texture fetch. For the depth peeling step, we use a depth texture to store the previous depth buffer. Before peeling a layer, we copy the current depth buffer into the depth texture using `glCopyTexSubImage2D`, and we then compare the clip-space (postperspective) depth zc_s of the incoming sample with the depth stored in the depth texture zc_{front}. We discard the samples for which $zc_s < zc_{front} + \varepsilon$, where $\varepsilon = 1.0^{-4}$. The ε is needed to account for the 24-bit precision of the depth buffer. The clip space coordinates of the incoming fragment are interpolated before doing the perspective division so that the interpolation is perspective correct.

For the search area optimization, we compute the hierarchical shadow map, using a shader. For every layer, the viewport width and height are divided by two. For every pixel, a depth value is computed by taking the *min* of the four depths of the associated subpixels in the upper level. This optimization, which was introduced by Guennebaud et al. [27], gives a 2x speed up for certain scenes on G70. However, on ATI R580, the optimization actually slows things down. We believe that this is because on R580, dynamic loops are more efficient than on G70, and so the numerous texture fetches from the hierarchical shadow map become the bottleneck.

5.5.2 Shadow Shader

Below is the Cg code of our shadow kernel, where *DepthMap* is the packed layered shadow map containing the four first depths for each light beam, and *BiasMap* are the depth gradients associated with the *DepthMap*, *kr* is the kernel radius in number of pixels described in Section 5.3.2, and p_{uv} is the projected pixel onto the near plane of the shadow frustum. This code is the bottleneck of the pipeline in all our results. Our shadow shader uses a dynamic double `for` loop to sample the search region, so it requires Shader Model 3 or higher.

```
half Area(half4 B)
{
     half2 V = B.yw - B.xz;
     half A = V.x * V.y;
     return A;
}

  //...
  half2 ij;
  for (ij.y = -kr; ij.y <= kr; ij.y++) {
    for (ij.x = -kr; ij.x <= kr; ij.x++) {

      half2 uv = p_uv.xy + ij * texel_width;
      half4 zs = tex2D(DepthMap, uv);
      half4 slope_bias = tex2D(BiasMap, uv);
      half4 midpoint_bias = (zs.yzww - zs) * 0.5;
      half4 biased_zs = zs + max(midpoint_bias, slope_bias);

      half4 occluding = (biased_zs < z) ? True : False;
      half4 r = zs * z / (z - zs);
      half4 V1 = c * (ij.xxyy + V0);
      half4 Bz = clamp(V1 * r.z, -0.5, 0.5);
      half4 By = clamp(V1 * r.y, -0.5, 0.5);
      half4 Bx = clamp(V1 * r.x, -0.5, 0.5);

      half A = (occluding.z) ? Area(Bz) : 0;
      if (A == 0) A = (occluding.y) ? Area(By) : 0;
      if (A == 0) A = (occluding.x) ? Area(Bx) : 0;
      I -= A;
    }
  }
```

CHAPTER 6

OTHER K-BUFFER APPLICATIONS

6.1 Depth Peeling Applications

6.1.1 Transparency

In real-time applications, transparency is usually simulated by compositing fragments in depth order, ignoring refraction at material interfaces. A common way to do this is to perform depth peeling to generate fragments in depth order and composite them into the framebuffer using α-blending [22]. For simplicity, we use a uniform α, but a nonuniform α can also be used. Figure 1.1 shows the results of single-pass transparency rendering using depth peeling with the k-buffer.

6.1.2 Translucency

Another application of depth peeling with the k-buffer is rendering translucency effects. We implemented a translucency algorithm that accounts only for absorption and does not simulate any scattering effects [59]. Assuming a bright ambient light, and ignoring reflection, translucency inside an homogeneous material can be rendered by computing an ambient term I_a using Beer-Lambert's law [61]: $I_a = e^{-\sigma_t l}$ where σ_t is the absorption coefficient and l is the distance that the light travels through the material. In certain cases, the thickness l can be computed in one pass without the k-buffer on current GPUs by summing the depths of the front and back faces separately using additive blending and taking the difference of the sums [26]. One advantage of depth peeling is that it can handle nonuniform σ_t terms. Indeed, the first k layers from the eye can be stored in the k-buffer and blended together using the volume rendering integral [38] instead of Beer-Lambert's law.

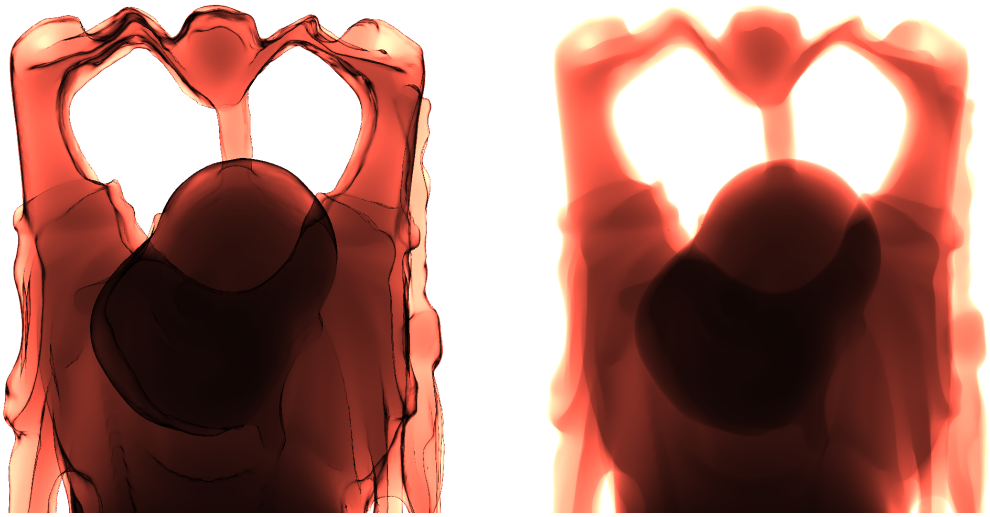
In reality, light rays are attenuated based on their incidence angle with the surface (Fresnel's effect). This effect is present for any type of material. For dielectric materials, it is common to use Schlick's approximation: $F_t = 1 - (1 - cos(\theta))^5$. These terms can be computed at each fragment and multiplied together. Figure 6.1 shows the result of depth peeling using the k-buffer, with and without Fresnel's effect.

6.1.3 Constructive Solid Geometry

Many complex shapes can be easily represented using constructive solid geometry (CSG). CSG operations on arbitrary objects can be represented as a boolean function, which is true for a point inside the new object and false otherwise. To render a CSG object with a boolean function, Kelley et al. [44] use a front-to-back depth ordering and encode the CSG function using a lookup table. With current programmable pixel shaders, the CSG function can be evaluated efficiently without the need of lookup tables. To perform CSG, we use our single-pass depth peeling to capture all the fragments of the scene. In the *k*-buffer, we store a linear depth and an object ID for each fragment. In a postprocessing pass, the fragments are traversed from front-to-back for each pixel. For each valid fragment, the state of the object (inside or outside) is updated. The first time the boolean function returns true, the depth of the fragment is stored and used in a deferred shading pass to construct normals and shade the object using central differencing [48]. Figure 6.2 shows the results of a CSG operation using the *k*-buffer.

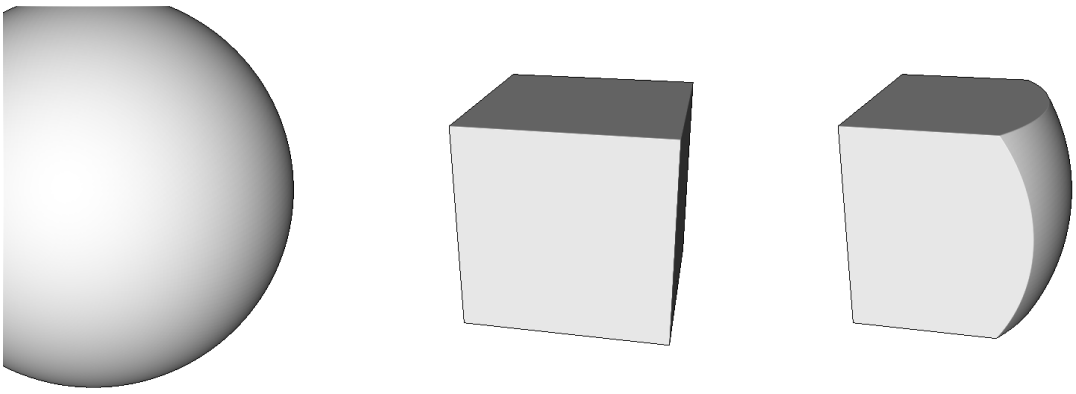
6.2 Depth Partitioning Applications

Similarly to depth peeling, the *k*-buffer can also be used to partition fragments into multiple depth ranges in a single rendering pass. Rather than storing the first k fragments along a ray like in depth peeling, depth partitioning keeps at most one fragment per depth partition. One effect that can take advantage of depth partitioning is blur-based depth-of-field, for which the background pixels may bleed onto the foreground. To avoid this bleeding, foreground fragments can be separated from the background fragments, keeping the nearest fragment to the eye in each partition. Using the *k*-buffer, up to k depth ranges can be captured in a single pass. This is done by comparing the depth of an incoming visible fragment with constants that define the ranges and placing the fragments in their correct range location in the *k*-buffer. Depth partitioning can also be used to voxelize a scene. Eisemann and Décoret use OpenGL bitwise logical operations to populate the voxels of a 3D grid [20]. These voxels are captured in a single pass, with one bit per voxel. The depth partition, called slice map, can be used to render effects such as opacity shadow maps and refraction. Besides, plausible soft shadows can be rendered efficiently by blurring the voxels in each depth partition [21]. Tariq and Llamas [68] use a different voxelization algorithm based on rendering to a 3D texture. They use the voxel grid to simulate and render smoke interacting with a moving manifold mesh. The 3D grid is ray traced in real time in a pixel shader. Their technique could be used to simulate other fluids, such as water.



(a) (b)

Figure 6.1. Translucency effects on the Happy Buddha (1,087,000 triangles) by depth peeling from the eye with a k-buffer. (a) Beer's Law with Fresnel's terms reflecting black ($k = 8$). (b) Same, without Fresnel's terms.



(a) (b) (c)

Figure 6.2. Example of a CSG (constructive solid geometry) operation using the k-buffer. (a) A = sphere, (b) B = cube, (c) $A \cap B$.

6.2.1 Depth of Field

Given depth partitions of the visible fragments, depth-of-field can be performed on the GPU by applying a depth-based blur that is weighted by the distance from the focal plane (i.e., a spatially-varying blur based on the circle-of-confusion). The drawback of this approach is that fragments from the background bleed onto the foreground and a sharp background cannot be seen behind blurry, transparent foreground objects. These problems are largely ameliorated by partitioning the scene into foreground, midground, and background depth layers [59, 46, 5], blurring each layer separately, and compositing them together. This approach requires rendering the entire scene three times with different near and far planes. With the k-buffer, we can route the foreground, midground, and background fragments into separate buffers based on their depth values. Figure 6.3 shows an example of depth-of-field using the k-buffer.

6.3 Sorting and Blending Applications

For effects such as transparency or volume rendering that require visibility ordering with arbitrary depth complexity, depth peeling a fixed number of layers may not be sufficient to render the effect properly. In case of overflow of the k-buffer, one can either merge fragments in the k-buffer [14, 41], or blend one fragment with the current color buffer [12]. This later blending approach assumes that the fragments are generated in a front-to-back, nearly-sorted order, e.g., sorting the primitives by the depth of their centroid.

To perform programmable blending with the k-buffer, a RMW (read modify write) framebuffer is required for compositing. For every fragment that is rasterized, the following steps take place. First, the k-buffer entries are read and compared along with the incoming fragment to find the two fragments closest to the eye (f_1 and f_2). A value such as color or depth is then computed using f_1, f_2, and the distance between them. This value is then composited into the framebuffer. Finally, the f_2 fragment along with the unused fragments are written back into the k-buffer and the f_1 fragment is discarded.

To ensure that the fragments are rasterized in a nearly-sorted visibility order, some object-space sorting is usually required. We perform the object-space sorting using a Least Significant Digit Radix Sort [63], which operates in linear time on floating point values with a simple float-to-int conversion [12]. The k-buffer finalizes the order in image-space by selecting the fragments closest to the eye from the k stored fragments. For scenes with many objects of low depth complexity, this object-space ordering can be accomplished by simply rendering these objects in depth order.

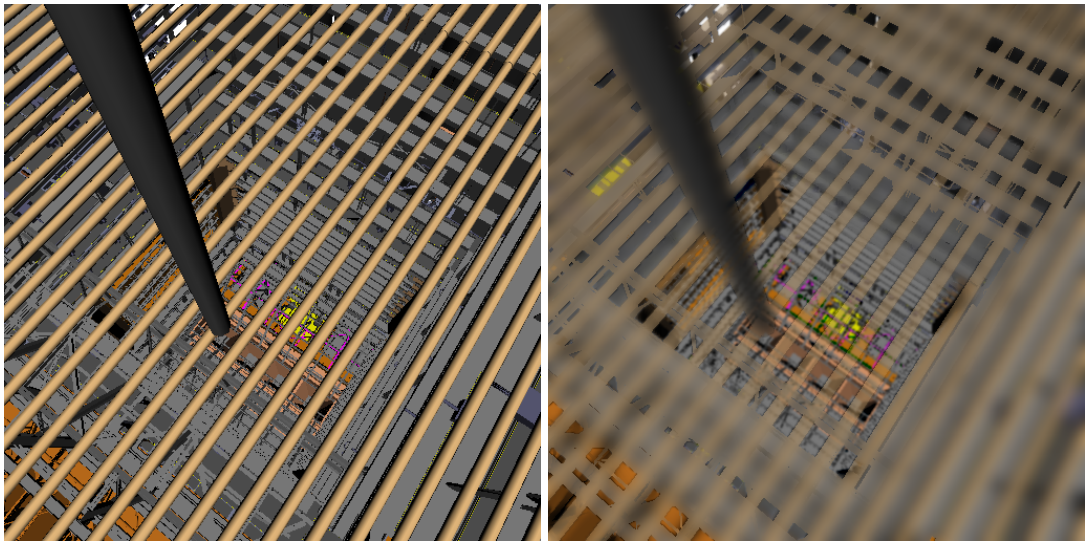
6.3.1 Isosurface Rendering

A texture-based approach for isosurface rendering of tetrahedral meshes was proposed by [72] which projects the tetrahedra in screen-space to triangles. The method uses a texture lookup to determine if interpolated texture coordinates correspond to an iso-value or not. Using the *k*-buffer, similar isosurface extraction can be performed directly on the triangles that compose the mesh. Our technique extracts the isosurface without the need to update a texture for each iso-value, and works with an arbitrary number of isosurfaces. For each fragment, the first and second nearest fragments to the eye are selected using the *k*-buffer. This forms a ray segment. If the iso-value is in the range of the current ray segment, then the depths of the fragments are linearly interpolated to find the depth of the isosurface on this ray segment, and the generated fragment goes through a depth test. (An entry of the *k*-buffer is used as a depth buffer.) To optimize the size of the *k*-buffer, we only store a depth value and a scalar value with each fragment. The normals are computed in a postprocessing pass using central differencing on the depths [48]. The surface is then shaded using a Lambertian term in eye space and silhouettes are computed (without additional cost).

This approach works equally well for tetrahedral meshes as well as for particles (points). To our knowledge, this is the first GPU-based approach for interactively extracting isosurfaces from particle data. Figure 6.4a and Figure 6.4b show the results of isosurface extraction from a tetrahedral mesh as well as from Material Point Method (MPM) simulation particles.

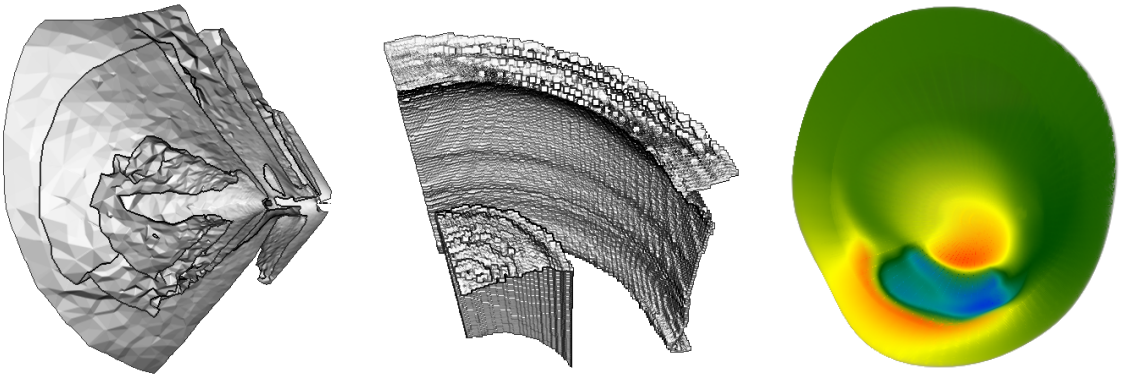
6.3.2 Volume Rendering

The first *k*-buffer application was for the specific case of direct volume rendering of unstructured grids [12]. Using programmable blending with the *k*-buffer, the ray gaps between triangles can be composited into the framebuffer for single-pass volume rendering. For each fragment, the *k*-buffer is used to find the two fragments closest to the eye. The scalar values of these two fragments, along with the distance between their depths, are used to look up the color contribution for the ray gap in a precomputed table of volume rendering integrals. This color is then composited in front-to-back order. See [12] for more detail. Figure 6.4c shows an example of volume rendering of a tetrahedral mesh using the *k*-buffer. Since this application uses a texture fetch (table lookup) in the *k*-buffer program, it would be supported only by the fragment-shader option and not by the blending option (*cf.* Section 4.3.1).



(a) (b)

Figure 6.3. Single-pass depth-range partitioning. Partitioning the fragments into foreground and background is necessary to render a sharp background underneath a blurry foreground. (a) Without depth-of-field (pinhole camera). (b) With foreground depth-of-field. The foreground, midground, and background are rendered into three separate images using an RGBZ *k*-buffer.



(a) (b) (c)

Figure 6.4. Volume visualization with the *k*-buffer. (a) Isosurface extraction of the Fighter tetrahedral mesh (1,403,504 tetrahedra). (b) Isosurface extraction of the Bullet007 MPM dataset (549k particles) with a constant point size. (c) Direct volume rendering of the Heart dataset using our *k*-Buffer extension of Mesa.

6.4 Results

For our experimental results, we rendered several large scenes using depth peeling with the *k*-buffer. We used our current experimental implementation in OpenGL based on RMW textures to demonstrate the optimal throughput of the *k*-buffer. The *k*-buffer attribute parameters were varied and a 512×512 viewport was used. To simulate a hardware *k*-buffer, we used OpenGL with GLSL shaders, 32-bit RGBA floating-point textures, and no Z culling. Our test machine was running Linux with an AMD Opteron at 2.2 GHz, 4 GB RAM, and an NVIDIA GeForce 7900 GTX with driver 1.0-8774. The scenes were rendered with three different modes: traditional multipass, single-pass with the *k*-buffer, and single-pass with the *k*-buffer including heuristics to decrease RMW artifacts. The last mode was used to generate most of our images and involved sorting all the triangles by their centroid on the CPU and batching them in sorted order with 32 triangles per batch. In all cases, our timing results represent the average framerates observed when rendering the scene without deferred shading (see Table 6.1). When the pipeline is vertex limited, we get a linear speedup with respect to geometry passes.

6.5 Discussion

Using a *k*-buffer to implement algorithms that operate on multiple fragments per pixel in a single geometry pass has two important benefits. First, for large datasets, each rendering pass of the geometry in the scene reduces the interactivity of the system substantially. Thus, effects that require multiple passes can greatly affect the usability of the system. With the *k*-buffer, this cost can be drastically reduced by capturing the relevant fragments in the first pass. Another important benefit of the *k*-buffer is that it simplifies the rendering of effects. In large rendering engines, each effect that is incorporated will add to the complexity of the code. With the *k*-buffer, all of the raster operations are encapsulated inside a single *k*-buffer shader, rather than having part of it controlled by fixed-function in the application and other parts controlled by a shader. This makes the shaders self-contained, and simplifies effect development.

A first step toward supporting a programmable *k*-buffer in GPUs could be a simple depth peeling raster operation for MRTs. This operation could be implemented as a set of depth and associated color buffers (i.e., multiple depth buffers). At each pass, these buffers would be filled in front-to-back order with the layered fragments using an insertion sort. This approach would be much simpler than a full *k*-buffer, because it does not require programmability.

Table 6.1. Timing results for depth peeling using traditional multipass rendering (MP), single-pass rendering with the *k*-buffer (SP), and single-pass rendering with the *k*-buffer using heuristics to avoid RMW hazards (SPwH). Several *k*-buffer layer sizes (4 or 16) and attribute combinations (RGBZ or Z) are compared.

k-Buffer	Dataset	Num Tris	MP	SP	SPwH
4 RGBZ	Dragon	871k	41.4 fps	139 fps	3.1 fps
16 Z			10.3 fps	139 fps	2.6 fps
4 RGBZ	Powerplant	12.7M	5.1 fps	20.1 fps	0.2 fps
16 Z			1.3 fps	20.1 fps	0.2 fps
4 RGBZ	Lucy	28.0M	0.4 fps	1.7 fps	0.2 fps
16 Z			0.1 fps	1.7 fps	0.1 fps

Instead, it would simply be enabled by the user in the API (e.g., `GL_DEPTH_PEELING`). It could also possibly allow early-Z tests by comparing with all the stored depth values. Since many of the effects described in this chapter can be performed with depth peeling, this change would have a high impact at a relatively low cost.

DirectX 10 [10] enables single-pass object-space depth partitioning by computing a render target index in a geometry shader. It can also perform fragment-level depth range culling differently for each render target using one viewport per render target. However, in this case, the geometry may need to be rasterized once per depth partition. The advantage of the *k*-buffer is that it needs the geometry to be rasterized only once.

CHAPTER 7

CONCLUSIONS

The first part of this thesis addressed the problem of capturing multiple fragments per pixel the most efficient way. First, we studied how to best send the geometry to the GPU, which is an important and general problem. We hope that this study will help GPU developers to render geometry more efficiently. In the second chapter, we designed a way to capture multiple fragments per pixel in a single geometry pass, which requires read-modify-write (RMW) fragment-level operations. We formulated a general data structure for storing and processing k fragments per pixel, called the k-buffer. We experimented with current hardware, and provided two possibilities of implementing RMW in future GPUs.

In the second part of this thesis, we presented a novel variation of soft shadow mapping built on depth peeling. While all shadow mapping algorithms until now have used one or two depth layers in their shadow maps, our algorithm uses an arbitrary number of depth layers (though three works well in practice), to help remove surface acne and light bleeding. Our algorithm is currently too slow to be used in games on current GPUs. However, it will become more practical on the next generation of GPUs (i.e., GeForce 8 and R600) which has more than four times the fragment shading power of the GeForce 7 generation. We closed the thesis with a chapter on other applications of the k-buffer, including transparency, translucency, constructive solid geometry, and depth of field.

We hope that this thesis will help fill in the gap between GPU experts and 3D application developers. High-level information about geometry feeding is broadly available, but results of microbenchmarks are rare. Our results should be useful when making decisions about what file formats to use to export 3D models, and how to condition the data, which is one of the goals of the COLLADA project [4].

In Chapter 4, we use the k-buffer read-modify-write data structure, which has been used for visualization of unstructured grids on the GPU [12] since 2005, and apply it to single-pass depth peeling [5]. Although it suffers from pipeline hazards, our implementation of the k-buffer is efficient and for cache-coherent meshes with small triangles, the artifacts may be

acceptable. We also believe that since the pipeline hazards are sometimes barely noticeable, a twist in the hardware to guarantee no read-modify-write pipeline hazards would be possible with low performance cost. The GeForce 8 GPUs supports fast shared memory shared by multiple thread processors, and slower uncached load-store global memory. Using a combination of these memory spaces, we think that the *k*-buffer can be implemented on GeForce 8 GPUs. We are interested in investigating how to implement hazard-free *k*-buffers on future GPUs, and in developing more multifragment effects based on the *k*-buffer.

In Chapter 5, we described incremental changes to a state-of-the-art soft shadow mapping technique based on backprojection of shadow-map fragments. We believe that our modifications for more robust self-shadowing and occluder fusion, are simple and incremental enough that they will be used in combination with other techniques. Layered soft shadow mapping is an example of application of the *k*-buffer, which could take advantage of it to generate a layered depth image in a single geometry pass. However, note that our current implementation does not use single-pass depth peeling because we did not want pipeline hazards from the *k*-buffer to impact the quality of our shadows. Anyway, single-pass depth peeling would have had no impact on performance on the small scenes that we used.

Applications shown in Chapter 6, such as transparency and thickness-based translucency are rarely used in games. We believe that part of the reason is that until now, these effects required multiple geometry passes. However, using the *k*-buffer, the effects can now be performed much more efficiently. We expect future games to have more transparency effects, using a *k*-buffer. However, effects which require rays to change direction, such as refraction or participating media, do not easily map to a *k*-buffer. For these effects, ray tracing is still the most accurate solution. Plausible soft shadows can be rendered from shadow maps, but for physically-accurate shadows, shadow volumes and ray tracing are currently the only correct classes of algorithms.

REFERENCES

[1] M. Agrawala, R. Ramamoorthi, A. Heirich, and L. Moll. Efficient image-based methods for rendering soft shadows. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 375–384, July 2000.

[2] L. Atty, N. Holzschuch, M. Lapierre, J.-M. Hasenfratz, C. . Hansen, and F. Sillion. Soft shadow maps: Efficient sampling of light source visibility. *Computer Graphics Forum*, 2006. (to appear).

[3] L. S.-K. Barnabs Aszdi. Real-time soft shadows with shadow accumulation. In *Eurographics 2006 - Short Presentations*. ACM, ACM Press, 2005.

[4] M. Barnes. Collada. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, page 8. ACM Press, 2006.

[5] L. Bavoil, S. Callahan, A. Lefohn, J. Comba, and C. Silva. Multi-fragment effects on the gpu using the k-buffer. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2007. (to appear).

[6] L. Bavoil, S. Callahan, and C. Silva. Robust soft shadow mapping with depth peeling. SCI Institute Technical Report UUSCI-2006-028, University of Utah, 2006.

[7] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. Scheidegger, C. Silva, and H. Vo. Vistrails: Enabling interactive multiple-view visualizations. In *IEEE Visualization '05*, pages 135–142, 2005.

[8] L. Bavoil and C. T. Silva. Real-time soft shadows with cone culling. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 105. ACM Press, 2006.

[9] F. F. Bernardon, J. L. D. Comba, C. A. Pagot, and C. T. Silva. GPU-based tiled ray casting using depth peeling. *Journal of Graphics Tools*, 11.3, 2006.

[10] D. Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[11] S. P. Callahan, L. Bavoil, V. Pascucci, and C. T. Silva. Progressive volume rendering of large unstructured grids. *IEEE Transactions on Visualization and Computer Graphics*, 12(5), September-October 2006.

[12] S. P. Callahan, M. Ikits, J. L. Comba, and C. T. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[13] D. Calver. Vertex decompression using vertex shaders. In *Direct3D ShaderX*. Wordware, 2002.

[14] L. Carpenter. The A-buffer, an antialiased hidden surface method. In *Proceedings of SIGGRAPH*, volume 18, pages 103–108, July 1984.

[15] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of Computer Science, University of Utah, 1974.

[16] H. Chong and S. J. Gortler. A lixel for every pixel. In *Proceedings of the 2nd EG Symposium on Rendering*, Springer Computer Science. Eurographics, Eurographics Association, 2004.

[17] M. Deering. Geometry compression. In *SIGGRAPH '95: Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, pages 13–20. ACM Press, 1995.

[18] M. Doggett. Xenos: Xbox360 gpu. *ATI*, 2005. Eurographics 2005 Slides.

[19] W. Donnelly and A. Lauritzen. Variance shadow maps. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 161–165, 2006.

[20] E. Eisemann and X. Décoret. Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 71–78, 2006.

[21] E. Eisemann and X. Décoret. Plausible image based soft shadows using occlusion textures. In *Proceedings of the Brazilian Symposium on Computer Graphics and Image Processing, 19 (SIBGRAPI)*, Conference Series. IEEE, IEEE Computer Society, 2006.

[22] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 2001.

[23] R. Fernando. Adaptive techniques for hardware shadow generation. Master's thesis, Cornell University, 2002.

[24] R. Fernando, S. Fernandez, K. Bala, and D. P. Greenberg. Adaptive shadow maps. In *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 387–390. ACM Press, 2001.

[25] N. K. Govindaraju, M. Henson, M. C. Lin, and D. Manocha. Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pages 49–56, 2005.

[26] S. G. Greg James. Real-time animated translucency. *NVIDIA Corporation*, 2004. (GDC 2004 Slides).

[27] G. Guennebaud, L. Barthe, and M. Paulin. Real-time soft shadow mapping by backprojection. In *Eurographics Symposium on Rendering*, 2006.

[28] T. Hachisuka. High-quality global illumination rendering using rasterization. In M. Pharr, editor, *GPU Gems 2*, chapter 38, pages 615–633. Addison Wesley, Mar. 2005.

[29] P. Haeberli and K. Akeley. The accumulation buffer: hardware support for high-quality rendering. In *SIGGRAPH '90: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, pages 309–318. ACM Press, 1990.

[30] J.-M. Hasenfratz, M. Lapierre, N. Holzschuch, and F. Sillion. A survey of real-time soft shadows algorithms. In *Eurographics*. Eurographics, Eurographics, 2003. State-of-the-Art Report.

[31] J. Hasselgren and T. Akenine-Möller. Efficient depth buffer compression. In *Graphics Hardware 2006*, Sept. 2006.

[32] P. S. Heckbert and M. Herf. Simulating soft shadows with graphics hardware. Technical Report CMU-CS-97-104, Jan. 1997.

[33] B. Heidelberger, M. Teschner, and M. H. Gross. Real-time volumetric intersections of deforming objects. In *VMV*, pages 461–468, 2003.

[34] H. Hoppe. Optimization of mesh locality for transparent vertex caching. In A. Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, pages 269–276, Los Angeles, 1999. Addison Wesley Longman.

[35] J.-C. Hourcade and A. Nicolas. Algorithms for antialiased cast shadows. *Computer & Graphics*, pages 259–265, 1985.

[36] M. Houston, A. J. Preetham, and M. Segal. A hardware F-buffer implementation. Technical Report CSTR 2005-05, Stanford University, 2005.

[37] R. Huddy. Graphics performance. *ATI*, 2006. (Slides).

[38] M. Ikits, J. M. Kniss, A. Lefohn, and C. D. Hansen. Volume rendering techniques. *GPU Gems*, pages 667–692, 2004.

[39] Y.-H. Im, C.-Y. Han, and L.-S. Kim. A method to generate soft shadows using a layered depth image and warping. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):265–272, 2005.

[40] G. S. Johnson, J. Lee, C. A. Burns, and W. R. Mark. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph.*, 24(4):1462–1482, 2005.

[41] N. P. Jouppi and C.-F. Chang. z^3: an economical hardware technique for high-quality antialiasing and transparency. In *SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 85–93, Aug. 1999.

[42] D. S. Kay and D. Greenberg. Transparency for computer synthesized images. In *SIGGRAPH '79: Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, pages 158–164. ACM Press, 1979.

[43] B. Keating and N. Max. Shadow penumbras for complex objects by depth-dependent filtering of multi-layer depth images. In *Proceedings of the 10th Eurographics Workshop on Rendering*, pages 205–220. Springer-Verlag, 1999.

[44] M. Kelley, K. Gould, B. Pease, S. Winner, and A. Yen. Hardware accelerated rendering of CSG and transparency. In *Proceedings of SIGGRAPH*, pages 177–184, 1994.

[45] E. Kilgariff and R. Fernando. The GeForce 6 series GPU architecture. In M. Pharr and R. Fernando, editors, *GPU Gems 2*, chapter 30, pages 471–491. 2005.

[46] A. E. Lefohn. *Glift: Generic Data Structures for Graphics Hardware*. PhD thesis, University of California Davis, 2006.

[47] B. Liu, L.-Y. Wei, and Y.-Q. Xu. Multi-layer depth peeling via fragment sort. Technical Report MSR-TR-2006-81, June 2006.

[48] Y. Livnat and X. Tricoche. Interactive point-based isosurface extraction. In *Proceedings of IEEE Visualization*, pages 457–464, 2004.

[49] B. Lloyd, D. Tuft, S. Yoon, and D. Manocha. Warping and partitioning for low error shadow maps. In *Proceedings of the Eurographics Symposium on Rendering 2006*, pages 215–226. Eurographics Association, 2006.

[50] T. Luft and O. Deussen. Real-time watercolor illustrations of plants using a blurred depth test. In *Proceedings of the 4th International Symposium on Non-Photorealistic Animation and Rendering*, 2006.

[51] A. Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9:43–55, July 1984.

[52] W. R. Mark and K. Proudfoot. The F-buffer: a rasterization-order FIFO buffer for multi-pass rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 57–64, 2001.

[53] T. Martin and T.-S. Tan. Anti-aliasing and continuity with trapezoidal shadow maps. In *Proceedings of the 2nd EG Symposium on Rendering*, Springer Computer Science. Eurographics, Eurographics Association, 2004.

[54] N. Max and K. Ohsaki. Rendering trees from precomputed Z-buffer views. In *Proceedings of the 6th Eurographics Workshop on Rendering*, 1995.

[55] A. Mendez, M. Sbert, J. Cata, N. Sunyer, and S. Funtane. Real-time obscurances with color bleeding. In *ShaderX4: Advanced Rendering Techniques*. Charles River Media, 2006.

[56] T. Moller and E. Haines. *Real-time rendering*. A. K. Peters, Ltd., 2002.

[57] Z. Nagy and R. Klein. Depth-peeling for texture-based volume rendering. In *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, 2003.

[58] D. Nehab, J. Barczak, and P. V. Sander. Triangle order optimization for graphics hardware computation culling. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 207–211, New York, NY, USA, 2006.

[59] NVIDIA. GPU programming exposed: The naked truth behind NVIDIA's demos. *NVIDIA Corporation*, 2005. (SIGGRAPH 2005 Slides).

[60] T. Ochotta and S. Hiller. Hardware rendering of 3d geometry with elevation maps. In *SMI '06: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2006 (SMI'06)*, page 10, Washington, DC, USA, 2006.

[61] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[62] C. Schler. Eliminate surface acne with gradient shadow mapping. In *ShaderX4: Advanced Rendering Techniques*. Charles River Media, 2006.

[63] R. Sedgewick. *Algorithms In C*, pages 298–301,403–437. Addison-Wesley, third edition, 1998.

[64] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast shadows and lighting effects using texture mapping. In *SIGGRAPH '92: Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, pages 249–252. ACM Press, 1992.

[65] J. Shade, S. Gortler, L.-W. He, and R. Szeliski. Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pages 231–242. ACM Press, 1998.

[66] P. Shirley and R. K. Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2003.

[67] H. Sowizral. Using a rendering pipeline efficiently. *SIGGRAPH'95 Course, No. 9*, 1995.

[68] S. Tariq and I. Llamas. Real-time volumetric smoke using d3d10. *NVIDIA Corporation*, 2007. (GDC 2007 Slides).

[69] J. E. Thornton. Parallel operation in the control data 6600. pages 32–39, 1964.

[70] D. Uesu, L. Bavoil, S. Fleishman, J. Shepherd, and C. T. Silva. Simplification of unstructured tetrahedral meshes by point sampling. In *Volume Graphics*, pages 157–165, 2005.

[71] Y. Wang and S. Molnar. Second-depth shadow mapping. Technical report, 1994.

[72] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):163–175, 2003.

[73] D. Weiskopf and T. Ertl. Shadow mapping based on dual depth layers. In *Proceedings of Eurographics '03 Short Papers*, pages 53–60, 2003.

[74] D. Wexler, L. Gritz, E. Enderton, and J. Rice. GPU-accelerated high-quality hidden surface removal. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 7–14, 2005.

[75] L. Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, volume 12, pages 270–274, Aug. 1978.

[76] M. Wimmer, D. Scherzer, and W. Purgathofer. Light space perspective shadow maps. In A. Keller and H. W. Jensen, editors, *Rendering Techniques 2004 (Proceedings of the Eurographics Symposium on Rendering 2004)*, pages 143–151. Eurographics, Eurographics Association, June 2004.

[77] C. Wittenbrink. R-Buffer: A pointerless A-buffer hardware architecture. In *ACM-Eurographics Workshop on Graphics Hardware*, pages 73–80, 2001.

[78] M. Wloka. Batching 4eva. *NVIDIA Corporation*, 2005. (GDC 2005 Slides).

[79] A. Woo. The shadow depth map revisited. In *Graphics Gems III*, pages 338–342. 1992.

[80] M. Woo and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1. 4*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[81] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. *ACM Trans. Graph.*, 24(3):886–893, 2005.