Shadow Techniques for Interactive and Real-Time Applications

Stefan Brabec

Max-Planck-Institut für Informatik
Saarbrücken, Germany

Dissertation zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Eingereicht am 18. Dezember 2003

Betreuender Hochschullehrer — Supervisor
Prof. Dr. Hans-Peter Seidel, MPI für Informatik, Saarbrücken, Germany

Gutachter — Reviewers
Prof. Dr. Hans-Peter Seidel, MPI für Informatik, Saarbrücken, Germany
Prof. Dr. Marc Stamminger, Universität Erlangen-Nürnberg, Erlangen, Germany

Dekan — Dean
Prof. Dr. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany

Datum des Kolloquiums — Date of Defense
3. 2. 2004

Abstract

Shadows provide important visual cues for the relative position of objects in three-dimensional space. For interactive and real-time applications, e.g. in virtual reality systems or games, the shadow computation needs to be extremely fast, usually synchronized with the display's refresh rate. Using dynamic scenes with many, movable light sources, shadow computation is therefore often the main bottleneck in a rendering system.

In this thesis we will discuss this problem in detail: Originating from Williams' shadow maps and Crow's shadow volumes, we will present hardware-accelerated shadow techniques that are able to generate shadows of high-quality while still being fast enough to be used in real-time or interactive applications. We will show algorithms for the computation of hard shadows as well as for the more complex problem of approximating soft shadows caused by area light sources.

Kurzfassung

Schatten sind wichtige visuelle Merkmale die über die relative Position eines Objektes in einem drei-dimensionalen Raum Aufschluss geben. Die Schattenberechnung muss für interaktive und Echtzeit-Anwendungen, wie z.B. Virtual Reality Systeme oder in Spielen, extrem schnell erfolgen, idealerweise synchronisiert mit der Bildwiederholfrequenz. Im Fall von dynamischen Szenen mit vielen, beweglichen Lichtquellen, ist die Berechnung von Schatten oftmals der zeitkritischste Teil innerhalb eines Rendering-Systems.

In dieser Dissertation behandeln wir genau dieses Problem im Detail. Ausgehend von Williams' Shadow Maps und Crow's Shadow Volumes werden Hardware-beschleunigte Schattentechniken vorgestellt, die Schatten von hoher Qualität erzeugen können, aber trotzdem so effizient sind, dass sie für Echtzeit- und interaktive Anwendungen eingesetzt werden können. Wir werden sowohl Algorithmen zur Berechnung harter Schatten beschreiben, als auch das schwierigere Problem der Approximation von sanften Schatten, wie sie z.B. bei Flächenlichtquellen entstehen, behandeln.

Summary

The accurate and efficient computation of shadows is one of the major challenges in the context of digital image synthesis. Today, a vast number of techniques exist that are able to produce very realistic shadows caused by complex light sources. But since the main focus of these algorithms is the quality of the resulting shadows, implementation and efficiency aspects are often neglected. As a consequence, these software-based techniques are often not able to perform at the high image refresh rates of current real-time, hardware-accelerated rendering applications.

In this dissertation we will focus on exactly this problem: computing realistically looking shadows for fully dynamic scenes using graphics hardware. Given this problem statement, we have to deal with three important requirements when developing a real-time shadow algorithm.

First, the resulting shadows should look realistic. Although physically correctness is the major goal in digital image synthesis, it is often sufficient to have a rough approximation that looks visually pleasing.

Second, we can not make any assumptions about the input data to process. Since most real-time applications are coupled with human interaction, we normally have no prior knowledge of spatial relationship or movement of lights and objects. The shadow algorithm therefore has to be general and robust enough to handle all possible configurations.

Third, the design of the algorithm needs to be tailored to the underlying graphics hardware. This is often one of the hardest requirements to fulfill: The pipeline architecture of current graphics systems is specialized for fast, parallel processing of small data packets, such as triangles or pixels. Since shadows are a global effect, based on the spatial relationship of objects and lights in the scene, we need to find methods of storing and accessing global scene information that work in this pipeline model.

In this thesis we present several shadow techniques that have been designed based on these three criteria. The first two parts focus on methods to compute shadows for point light sources which are extensions of the shadow mapping (first part) and shadow volumes (second part) techniques. In the third part, we propose two approaches for the more complex problem of computing soft shadows caused by extended light sources, such as linear or area lights.

Shadow Mapping

In the first part we present several extensions to the classical shadow mapping technique. Shadow mapping can be seen as a special variant of texture mapping and therefore perfectly fits to the design of current graphics hardware. But since it is an image-based method, we also have to deal with sampling artifacts that result, e.g., from a low sampling density or numerical precision issues.

First, we present a set of algorithms that can be used to reduce these visible artifacts. The main idea here is to first analyze the spatial arrangement of camera,

light source, and scene objects and use this information to optimize the shadow map settings. This way, most of the available precision is used for the visible parts in the scene, rather than wasting samples for something that is not in the actual view of the camera.

Next, we propose a shadow map filtering method that can be used for hardware-accelerated shadow mapping without the need of dedicated hardware functionality. Based on percentage closer filtering, we show how to map this filtering scheme to hardware by color-coding depth values and applying image-processing techniques, for which hardware-accelerated architectures exist.

The next chapter in this part of the thesis deals with the problem of using shadow maps for light sources with hemispherical or omnidirectional characteristics. Since the classical shadow mapping method is based on a perspective projection, a single shadow map can only capture a relatively small field-of-view but not the complete environment. We present a solution to this problem by replacing the perspective projection with a dual-paraboloid mapping. Dual-paraboloid shadow mapping reduces the number of shadow maps needed to capture the surrounding environment and is also suitable for hardware implementation.

The last algorithm presented in this part concentrates on performance aspects of shadow mapping. In many applications, the generation of shadow maps is implemented as an additional rendering pass that is solely used to obtain depth values of objects nearest to the light source. This means that during shadow map generation much of the hardware capabilities are not used. Therefore we propose the use of extended light maps, a combination of light maps and shadow maps. During shadow map generation we not only compute depth values, but also pre-calculate parts of the local illumination model for the given light source.

Shadow Volumes

The second part of this thesis concentrates on shadow volume techniques used for hardware-accelerated rendering.

First, we present an implementation of shadow volumes for application in an interactive, hybrid rendering system. By using graphics hardware to compute the direct illumination, we can dedicate more CPU resources for the expensive indirect illumination computation. However, for physically correct rendering we have to ensure that the direct illumination is computed at very high quality, including accurate shadow information, and also at very high speed. In this chapter we show how to efficiently compute shadows using the shadow volume algorithm for a number of light sources. To speed up the computation for multiple light sources, we combine the results of up to four lights in one temporary intensity texture, which is later mapped onto the final scene. This intensity texture is not only used to reduce the number of rendering passes needed for the scene, but can also be utilized to integrate non-uniform directional power distribution.

In the second chapter of this part, we present a fully hardware-accelerated shadow volume implementation that requires no CPU work at all and furthermore supports fully dynamic scenes, including object deformation. An expensive part of the shadow volume algorithm is the determination of silhouette edges for each shadow casting object. Silhouette edges are needed to generate the side planes of the corresponding shadow volumes. We show how to map this processing step to hardware by using floating point vertex and pixel processing units, available on newer graphics hardware. This way, the hardware can process vertex data at nearly the same precision as the CPU, but at a much higher speed. Apart from efficiency, one major aspect of our algorithm is that it can also detect silhouette edges for meshes which are deformed on the graphics hardware. This was previously only possible by simulating the hardware-based deformation in software, which is not only very inefficient but also very complicated to implement.

Soft Shadows

The third part of this thesis is dedicated to the computation of soft shadows caused by area or linear light sources. We present two techniques that approximate soft shadows using a relatively small number of samples on the light source itself, so that the algorithms can be used in a hardware-accelerated rendering system.

The first method works for linear light sources, as used, e.g., in architectural scenes to represent neon lamps. Here we propose a new shadow map variant, called soft shadow map. Soft shadow maps not only contain qualitative shadow information (lit or blocked) but also quantitative information (amount of energy arriving). This is achieved by first generating a standard shadow map for each sample point on the light source. Next, these maps are used to detect shadow boundaries as seen by each of the sample points. From these silhouette edges we can compute the quantitative information for the neighboring sample points. Since our method is mostly based on image processing and texture mapping, it is an ideal candidate for a hardware-accelerated implementation. Furthermore, we will show how to use soft shadow maps for approximating shadows caused by area light sources.

Next, we propose an algorithm that approximates soft shadows only using information obtained from a single sample point. As in the previous technique, the basis of our method is the traditional shadow mapping approach. However, instead of testing only a single entry in the shadow map, we search the neighborhood of a given sample to obtain information about the spatial arrangement of the scene. The results of this step are a number of distances, e.g. the distance to the next entry which is in shadow or lit, from which we calculate a penumbra value. Although this approach is far from being physically correct, it resembles most of the soft shadow characteristics and can also be implemented very efficiently.

Zusammenfassung

Die genaue und effiziente Berechnung von Schatten ist eine der Hauptherausforderungen im Bereich der digitalen Bildsynthese. Es existiert heutzutage eine beträchtliche Anzahl von Techniken, die in der Lage sind, sehr realistische Schatten wie sie durch komplexe Lichtquellen entstehen zu berechnen. Da aber das Hauptaugenmerk dieser Algorithmen auf der Qualität der berechneten Schatten liegt, werden Implementierungs- und Effizienzaspekte häufig vernachlässigt. Demzufolge sind jene Software-basierten Techniken häufig nicht in der Lage die Geschwindigkeitsanforderungen von gegenwärtigen Hardware-beschleunigten Echtzeit-Renderern zu erfüllen.

In dieser Dissertation konzentrieren wir uns auf genau dieses Problem: Die Hardware-unterstützte Berechnung von realistisch wirkenden Schatten für dynamische Szenenbeschreibungen. Aufgrund dieser Problemstellung müssen wir uns bei dem Entwurf eines Echtzeit-Schattenalgorithmus mit drei wichtigen Anforderungen beschäftigen.

Zunächst sollten die resultierenden Schatten natürlich aussehen. Obgleich eine physikalisch korrekte Berechnung das Hauptziel der digitalen Bildsynthese ist, ist es oftmals ausreichend, nur eine grobe Näherung zu erzeugen, die visuell richtig erscheint.

Die zweite Anforderung besteht darin, den Schattenalgorithmus von der zu berechnenden Szene zu trennen. Da die meisten Echtzeitanwendungen durch die Interaktion mit dem Benutzer gesteuert werden, können wir im Algorithmus grundsätzlich keine Annahmen über die Art der Szene oder die jeweilige Positionen der Objekte voraussetzen. Der Schattenalgorithmus muss deshalb so entworfen werden, dass eine zuverlässige Berechnung von allgemeinen Szenen in jeglicher räumlicher Konfiguration möglich ist.

Drittens muss das Design des Algorithmus auf die zugrundeliegende Graphikhardware zugeschnitten werden. Dies ist häufig eine der schwierigsten Anforderungen: Die Pipeline-Architektur, auf der viele Graphiksysteme basieren, ist für die schnelle, simultane Verarbeitung von kleinen Datenpaketen wie Dreiecken oder Pixeln spezialisiert. Da aber Schatten ein globaler Effekt sind, dem die räumliche Anordnung der Objekte und Lichtquellen zugrunde liegt, müssen wir jeweils Methoden entwickeln, die bezüglich Datenspeicherung und Datenabfrage in diesem lokalen Pipeline-Modell möglich sind.

In dieser Dissertation werden wir einige Schattentechniken vorstellen, deren Design auf diesen drei Anforderungen basiert. In den ersten beiden Teilen konzentrieren wir uns auf Verfahren, die Schatten für Punktlichtquellen berechnen. Dies sind zum einen Methoden die auf dem Schattenkartenalgorithmus basieren (erster Teil), zum anderen solche, die eine Erweiterung des Schattenvolumenansatzes darstellen (zweiter Teil). Im dritten Teil befassen wir uns mit dem komplizierteren Problem der Berechnung von weichen Schatten (Soft Shadows), wie sie z.B. durch Flächenlichtquellen entstehen. Hier werden wir zwei Verfahren vorstellen, die diese Art von Schatten näherungsweise berechnen.

Schattenkarten

Im ersten Teil stellen wir einige Erweiterungen und Verbesserungen der klassischen Schattenkartentechnik (Shadow-Mapping) dar. Dieses Verfahren kann als eine spezielle Variante des Texture-Mappings angesehen werden und eignet sich deshalb hervorragend für den Einsatz auf Graphikhardware. Da es allerdings eine bild-basierte Methode ist, müssen wir uns insbesondere mit Sampling-Problemen, die z.B. aufgrund einer zu geringen Auflösung der Schattenkarte auftreten, beschäftigen.

Zuerst befassen wir uns mit einer Reihe von Ansätzen, die verwendet werden können, um diese sichtbaren Sampling-Probleme zu verringern. Der Grundgedanke hier ist, dass wir zunächst die räumliche Anordnung von Kamera, Lichtquelle und Objekten der Szene analysieren und diese Informationen dazu nutzen, den Shadow-Mapping-Prozess zu optimieren. Auf diese Weise wird ein möglichst grosser Bereich der Schattenkarte für die sichtbaren Teile der Szene verwendet, anstatt Teile der Schattenkarte für von der Kamera aus nicht sichtbare Regionen zu vergeuden.

Als nächstes befassen wir uns mit Filterungs-Methoden die für das Hardware-unterstützte Schattenkarten-Verfahren geeignet sind. Ausgehend von der sogenannten Percentage-Closer-Filterung zeigen wir, wie dieses Verfahren auf die Graphikhardware übertragen werden kann. Hierbei gehen wir nicht davon aus, dass diese Methode direkt von der Hardware unterstützt wird, sondern nutzen die Standard-Funktionalität der Graphikkarte. Durch Farbkodierung der Tiefeninformationen kann das Filtern durch einfache Bildverarbeitungs-Operationen durchgeführt werden, die auf einigen Graphiksystemen von der Hardware unterstützt werden.

Die Verwendung des Schattenkartenverfahrens für Lichtquellen mit hemisphärischer oder omni-direktionaler Abstrahlcharakteristik ist der Inhalt des folgenden Kapitels. Da Shadow-Mapping üblicherweise auf einer perspektivischen Abbildung basiert, kann eine einzige Schattenkarte nur einen verhältnismässig kleinen Teil, aber nicht die gesamte Umgebung, repräsentieren. Aus diesem Grund stellen wir einen Ansatz vor, bei dem die perspektivische Projektion durch das sogenannte Dual-Paraboloid-Mapping ersetzt wird. Im Vergleich zur perspektivischen Abbildung reduziert sich durch diesen Algorithmus, der sich sehr gut für eine Hardware-basierte Implementierung eignet, die Anzahl der benötigten Schattenkarten erheblich.

Der letzte Algorithmus in diesem Teil behandelt Effizienzaspekte des Shadow-Mappings. In vielen Anwendung stellt die Generierung der Schattenkarte einen zusätzlichen Rendering-Durchlauf dar, der ausschliesslich der Berechnung der Tiefenwerte dient. Dies heisst, dass während dieser Phase ein Grossteil der Graphikkarten-Funktionalität nicht genutzt wird. In diesem Kapitel stellen wir deshalb das Konzept der Extended-Light-Maps – eine Kombination aus Light-Maps und Shadow-Maps – vor. Hierbei werden während der Generierung nicht nur die Tiefenwerte, sondern auch Teile der lokalen Beleuchtung berechnet.

Schattenvolumen

Der zweiten Teil konzentriert sich auf Schattenvolumenverfahren (Shadow-Volumes), die für Hardware-beschleunigtes Rendering eingesetzt werden können.

Zunächst stellen wir eine Shadow-Volume-Implementierung vor, die für ein hybrides (Hardware/Software)-Rendering-System entworfen wurde. Durch Auslagerung der lokalen Beleuchtungsberechnung auf die Graphikhardware können mehr CPU-Ressourcen für die kostspielige indirekte Beleuchtungsberechnung zur Verfügung gestellt werden. Hierbei muss allerdings sichergestellt werden, dass die lokale Beleuchtung, einschliesslich Schatten, sowohl effizient als auch mit möglichst hoher Qualität berechnet wird. In diesem Kapitel zeigen wir, wie sich der Shadow-Volume-Algorithmus für mehrere Lichtquellen effizient implementieren lässt. Hierbei wird die Schatteninformation für bis zu vier Lichtquellen in einer temporären Intensitätstextur abgelegt, die dann erst später mit dem berechneten Bild kombiniert wird. Um die Qualität der lokalen Beleuchtungsberechnung zu erhöhen, demonstrieren wir, wie diese Intensitätstexturen verwendet werden können, um beispielsweise Lichtquellen mit ungleichmässigem Abstrahlverhalten zu repräsentieren.

Im zweiten Kapitel befassen wir uns ebenfalls mit der Implementierung von Shadow-Volumes. Der vorgestellte Algorithmus erlaubt es, Shadow-Volumes für dynamische Szenen, inklusive z.B. Deformation, rein Hardware-basiert zu berechnen. Ein aufwendiger Teil des Shadow-Volume-Verfahrens ist die Bestimmung der Silhouettenkanten, die für jedes schattenwerfende Objekt durchgeführt werden muss. Wir zeigen wie durch den Einsatz von programmierbaren Pixel- und Geometrie-Einheiten, die bei heutiger Graphikhardware in Gleitkomma-Genauigkeit arbeiten, auch dieser Teil des Verfahrens Hardware-basiert implementiert werden kann. Bezüglich der Genauigkeit der Berechnung steht dieses Verfahren einer Software-basierten Implementierung in nichts nach, ist aber deutlich effizienter. Ein weiterer Vorteil dieser Hardware-basierten Silhouetten-Bestimmung zeigt sich bei Objekten, deren Geometrie von der Hardware verändert (deformiert) wird. Um in diesem Fall die Silhouetten zu bestimmen, müsste eine Software-Implementierung die Deformation, wie sie von der Graphikhardware durchgeführt wird, simulieren, was nicht nur ineffizient, sondern auch schwierig zu implementieren ist. Bei unserer Hardware-Implementierung dagegen ist dies kein Problem, da sämtliche Geometrie auf der Hardware verarbeitet wird.

Weiche Schatten

Im dritten Teil dieser Dissertation widmen wir uns der Berechnung von weichen Schatten, wie sie durch lineare oder auch Flächenlichtquellen entstehen können. Wir stellen zwei Verfahren vor die diese sanften Schattenübergänge ausgehend von wenigen Abtastpunkten auf der Lichtquelle approximieren, so dass eine Hardware-basierte Implementierung möglich wird.

Der erste Algorithmus eignet sich insbesondere für lineare Lichtquellen, wie sie z.B. häufig in Architekturszenen zur Simulation von Neonröhren eingesetzt werden. Wir benutzen hierzu sogenannte Soft-Shadow-Maps, die nicht nur qualitative Information (beleuchtet oder im Schatten), sondern auch quantitative Information (Menge der einfallenden Energie) enthalten. Bei der Generierung dieser Soft-Shadow-Maps wird zunächst eine herkömmliche Schattenkarte für jeden Abtastpunkt auf der Lichtquelle generiert. Anschliessend können wir anhand dieser Schattenkarten die Schattengrenzen aus Sicht jedes Abtastpunktes ermitteln und diese Grenzen den benachbarten Abtastpunkten als weiche Schattenübergänge zuordnen. Da unsere Methode hauptsächlich auf Bildverarbeitungsoperationen und Texture-Mapping beruht, eignet sie sich sehr gut für eine Hardware-basierte Implementierung. Desweiteren erklären wir, wie Soft-Shadow-Maps auch für Flächenlichtquellen eingesetzt werden können.

Ein zweiter, auch auf Schattenkarten basierender Algorithmus erlaubt die Approximation von weichen Schatten ausgehend von nur einem Abtastpunkt der Lichtquelle. Anstatt nur jeweils einen einzigen Eintrag in der Schattenkarte zu testen, analysieren wir zusätzlich die nähere Umgebung diese Eintrags, um Information über die räumliche Anordnung der Objekte zu erhalten. Das Resultat dieser Analyse ist eine Anzahl von Abständen, z.B. der Abstand zu dem nächstliegenden Eintrag der im Schatten liegt, mit deren Hilfe wir die quantitative Sichtbarkeit der Lichtquelle approximieren. Obgleich dieses Verfahren eine sehr grobe Näherung darstellt, die nicht physikalisch basiert ist, erzeugt es dennoch realistisch wirkende weiche Schattenübergänge und kann desweiteren sehr effizient implementiert werden.

Acknowledgements

I would like to acknowledge my supervisor, Professor Hans-Peter Seidel, for his guidance and support during the last years, not only during my stay at the MPI, but also throughout my studies, since my early days in computer graphics as a student at the University of Erlangen-Nuremberg. I would like to thank him for introducing me to the world of computer graphics and providing an excellent environment for my research.

I am also very grateful to Professor Marc Stamminger for becoming an external reviewer of this thesis.

Especially I like to thank Professor Wolfgang Heidrich for many discussions related to the field of hardware-accelerated rendering and computer graphics techniques in general. The work with him aroused my interest in graphics hardware and hardware-related programming and OpenGL tricks.

I also owe thanks to Karol Myszkowski for proof-reading early versions of this thesis and providing valuable remarks and improvements. Special thanks also to Thomas Annen, who did most of the implementation for the algorithms in Chapter 4 and Chapter 6.

This thesis would not have been possible without the enormous support of my colleagues of the computer graphics group at the MPI throughout the last four years. Especially I would like to thank the following people (in alphabetical order): Kirill Dmitriev, Michael Goesele, Jan Kautz, Hendrik Lensch, Christian Rössl, Annette Scheel, Philipp Slusallek, Hartmut Schirmacher, and Jens Vorsatz.

Thanks also go to the developer support teams of ATI, especially Michael Doggett, and NVIDIA for providing early drivers, programming info, and hardware samples.

Finally I would like to thank Katja and my family for their support during the years of this thesis.

Contents

Chapter 1
Introduction

Darkness is the absence of light. Shadow is the diminution of light. Primitive shadow is that which is attached to shaded bodies. Derived shadow is that which separates itself from shaded bodies and travels through the air. Repercussed shadow is that which is surrounded by an illuminated surface. The simple shadow is that which does not see any part of the light which causes it. The simple shadow commences in the line which parts it from the boundaries of the luminous bodies.

Leonardo da Vinci (1452 to 1519)

The study of the behavior of light and its effect in the surrounding environment has been, and is still nowadays, a challenging field of research. The above citation is an excerpt from Leonardo da Vinci's notebooks [Leonardo da Vinci45] where Leonardo has sketched his observations on the interaction of light and objects. He was one of the leading minds during the Italian Renaissance (1420-1600), in which artists started to observe the real world and came up with rules for applying light and shadow in a very natural way. The main stylistic element that was characteristic for that period was *chiaroscuro*[1]. The arrangement of light and dark regions in an image was extensively used to produce the illusion of depth in paintings. The rules of properly shading an object were defined by identifying different regions on the object and the surrounding environment. The impression of a glossy object was achieved by drawing highlights at that part of the object where the light source is most dominant, whereas the overall lit parts where drawn as the combination of object and light color, with decreasing intensity as the light source influence becomes less dominant. To emphasize the object's position in the scene, cast shadows are applied for the surrounding environment. Self-shadowing of an object was defined as a continuous color fade from a dark color (shadow) to black (core shadow). With more than one light source in the scene, an artist could further improve the three dimensional effect by adding reflected light to shadow and core shadow regions.

[1]from the italian works ”chiaro” (clear or light) and ”oscuro” (obscure or dark)

History in computer graphics made a quite similar way. In the early days, graphical output was mostly in wire frame mode and used for engineering applications, e.g., visualizing machine parts. During the 1960s, realism went a step further by having objects displayed as shaded solids, done with half-toning, pattern-based approaches similar to artistic pencil drawings techniques. This way depth and spatial relationship in computer generated images was enhanced. At the end of the sixties Arthur Appel presented his work on the shaded rendering of solids [Appel68]. This was probably one of the first publications on shadow techniques in the history of computer graphics. In this paper, different ways of shading solid objects, including shadows, were presented: A brute force, point by point shading approach but also optimizations, e.g., detecting contour edges relevant for shadow boundaries. With Appel's work the importance of shadows in computer generated images was emphasized: Shadows are important visual cues that make the spatial relationship of objects easier to understand.

In these early days of computer graphics, rendering an image took from several minutes up to hours or days, even for very trivial scenes. Algorithms like ray tracing or radiosity pushed realism to photo-realistic quality, but computation time was so enormous that no interaction was possible.

This changed immediately by the introduction of hardware-accelerated rendering. Companies like SGI developed powerful systems that had dedicated hardware support for lighting, texturing, and hidden surface removal. Although the amount of realism produced by hardware-accelerated rendering could not compete with offline methods of that time, the systems were able to produce images in a fraction of a second. This opened a new branch for computer graphics, called real-time rendering.

Using rasterization based graphics hardware involves a trade-off between quality and speed. One key concept that allows extreme parallel processing and fast rendering is the restriction to local illumination and simple reflection models. In contrast to global illumination, the appearance of an object depends only on a small number of parameters, e.g. the light source's position and direction, viewer, and surface material. Adding shadows to this type of architecture is difficult. Polygons are processed independently of each other, but shadow computation is based on the global arrangements of objects and light sources in a scene.

Two kind of algorithms are suitable to solve the global shadow task. One are the so called off-line methods which classify the shadow regions in a preprocessing step and only use the graphics hardware to visualize the final result. A popular method in this category are light maps generated from a global illumination system which are then applied as surface texture maps. In terms of quality this approach is one of the most accurate shadow techniques, if texture resolution is sufficiently high enough so that rasterization artifacts can not be seen. However, computing an accurate global illumination solution is an expensive, time-consuming task, so in terms of speed this approach is only suitable for static environments, e.g. architectural visualization where interaction is completely restricted to the change of viewing parameters.

For dynamic environments with changing objects and illumination, global shadows need to be updated at ideally the same rate as the display device. Interactive and real-time applications require updates of about 10 up to more then 60 frames per second, which puts a heavy load on the shadow computation. Furthermore, there are also special applications, such as virtual studios or simulation systems, which require a fixed output frame rate, regardless of the scene's complexity or current viewing parameters.

Shadow algorithms therefore have to trade-off between quality and speed: Shadows need to be visually pleasing but also computed at a fraction of a second, even for complex environments.

In addition to the requirement of handling fully dynamic environments efficiently we further specify the characteristic of the shadow algorithms discussed in this thesis by three more terms:

Realistic

As already mentioned before, the quality of the resulting shadows is among the most important properties of any shadow technique. The generated shadows should always correspond to the spatial relationship of objects in the scene and the type of light source being used. Especially in the context of real-time rendering, photorealistic, physically correct shadows are often not possible due to computation time restrictions. For most applications it is sufficient to have a shadow approximation that looks visually pleasing.

General

A shadow algorithm should make only few to no assumptions about the scene description itself or other parameters (camera, animation paths, etc.). Shadows should be of the same quality, putting no restrictions to the processed scene.

Hardware-accelerated but flexible

The shadow technique should make use of as much hardware-assistance as possible. This is not only a requirement in order to be able to handle very complex scenes at reasonable frame rates, but also in terms of resource management. Since the CPU is more and more dedicated to non-graphic work, e.g. sound processing or numerical simulation, CPU-based computation must to be minimized. In addition to this, the algorithm should be easy to integrate into an existing interactive rendering system. The cost of implementation and required changes to the core functionality of the system should therefore be minimized.

Designing suitable algorithms is a process of finding a reasonable trade-off between all these criteria. A fully hardware-accelerated method for example may only work for a specific class of scene objects, while a more hybrid approach, that involves much more CPU based computation, would support all types of objects.

An application developer should therefore carefully specify which requirements are most important and which can be restricted to special cases. Such a special case could be the type of light sources that are supported, e.g. an algorithm that computes shadows for spot lights with a limited cut-off angle can be implemented quite efficiently in contrast to shadow algorithms that support more general light source types.

The algorithms proposed in this thesis are all examples of emphasizing one or more criteria, as in the light source type example. Given the requirements the resulting shadow quality is then often dominated by the clever use of available resources, like CPU time or hardware-capabilities.

1.1 Contribution and Overview

The reminder of this thesis is organized as follows. The next chapter serve as an introduction to digital image synthesis and hardware-accelerated rendering techniques. This includes basic methods of computing visible surfaces as well as an introduction to reflection models, light sources, and shadows. And the end of the chapter we will also show how these concepts are realized on modern graphics hardware architectures.

The third chapter focuses on related work in the context of shadow algorithms. Here we will describe the major shadow algorithms that are suitable for efficient shadow computation in a real-time environment. Since there exists a huge number of publications in this area, we will concentrate on those being widely used in applications like games and virtual reality and those relevant for our work. This chapter also contains a detailed description of shadow maps [Williams78] and shadow volumes [Crow77] since many of our proposed algorithms are extended variants or special implementations of these techniques.

In the first part of this thesis we present several enhancements to Williams' shadow map technique for point light sources that can greatly improve shadow quality and rendering speed:

- A method that adjusts the light source's viewing frustum in order to reduce sampling artifacts [Brabec02a].
- An hardware-accelerated method for shadow map filtering [Brabec01] that can be implemented on standard OpenGL hardware.
- A specialized shadow map parameterization for hemispherical or omnidirectional light sources [Brabec02b].
- A combined light map / shadow map approach [Brabec00] that is capable of saving valuable hardware-resources.

The second part describes two approaches that are based on the shadow volume algorithm:

- A shadow volume implementation for complex environments that can be used for many light sources and special light source characteristics [Dmitriev02].

- A full hardware-accelerated implementation of the shadow volume approach, including silhouette detection and extraction [Brabec03].

In the third part, two novel algorithms are presented that can be used to create realistic shadows caused by area or linear light sources:

- A hardware-accelerated soft shadow technique for linear light sources [Heidrich00].
- A hybrid-method that approximates soft shadows using only a single shadow map [Brabec02c].

We will conclude this thesis and discuss future work in Chapter 12.

Chapter 2
Background

In this chapter we will describe some of the concepts needed for understanding the algorithms and methods presented in the following chapters of this thesis. A more detailed and complete overview of computer graphics techniques can be found in [Foley96]. A very extensive overview of the techniques used in digital image synthesis is presented in [Glassner95].

We start with a short introduction to hidden-surface removal techniques. The determination of visible (or hidden) surfaces is one of the major tasks when generating a digital image. As we will see in the next chapters, there is a duality between the computation of visible surfaces seen from the camera and the determination of lit and shadowed surfaces as seen from the light source.

Next, we introduce some of the lighting and shading models used in computer graphics today. Since there exists a vast number of different techniques, we will only focus on those relevant for real-time rendering. A major part of this section is the discussion of shadows caused by various types of light sources.

The last section deals with the architecture of graphics hardware and the principles of hardware-accelerated rendering. We will review the main parts of the general graphics pipeline but also discuss recent trends and features of graphics hardware.

2.1 Hidden-Surface Removal

One of the fundamental tasks in the process of digital image synthesis is the determination of visible surfaces for a given view and scene. Given a set of 3D opaque, solid objects only parts of the scene can be seen from a given point of observation. Objects far away may be completely hidden from the viewer by objects in between, whereas cases exists were only a fraction of an object may be visible.

Although the basic task sounds quite simple, one can imagine that as the number of objects in the scene or the final image resolution increases, the exact and efficient visible-surface determination gets more complicated.

According to [Foley96], there exist two fundamental approaches to the specific problem: For a given pixel in the final image, we can find the object that is closest to the viewer. Or, we can test each object in the scene, and try to find those parts of the object that are not blocked by any other object in the scene or parts of the object itself.

In this section we give a brief overview of algorithms for both classes, one is the *z-buffer* approach [Catmull75] that is the common visibility method used for hardware-accelerated rendering today, while the other is a list-priority algorithm that resolves visibility by sorting and splitting objects according to their spatial position [Newell72]. A comprehensive overview of visibility techniques can be found in [Foley96, Sutherland74].

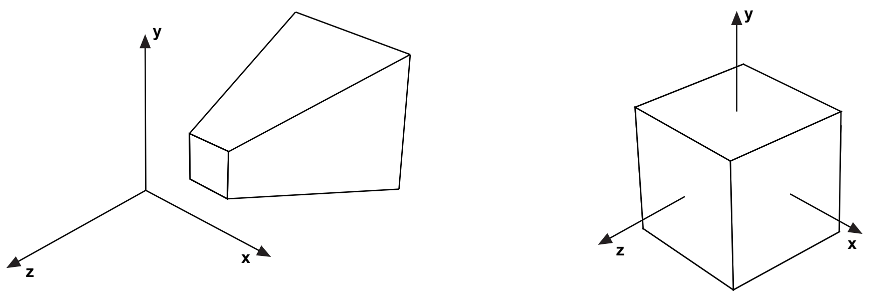
2.1.1 View Transformations

Since visibility is computed according to the actual camera setting the type of view transformation or camera model needs to be taken into account. Most rendering systems support two basic types of camera models, one is the classical pinhole camera model, the other is the directional camera, which is mostly used in CAD/CAM applications for technical illustration. There exist of course more sophisticated models that are more physically motivated, e.g. the thin lens model [Heidrich99].

A directional camera, or orthographic projection, is by far the most simplest model to implement. For a given surface point the point is projected orthogonally along the line of sight onto the image plane. It can easily be seen that this model is not really a good choice for realistic image synthesis, since distances are not taken into account; all objects are just *flattened* onto the image plane.

A better approach is the pinhole camera model. Here it is assumed that the camera itself is a simple box with an infinitesimal small hole on one side and a film on the opposite side of the box. Light falling through the hole will be projected as an upside-down image onto the film. Geometrically, this is nothing more than a perspective projection using the hole as the center of projection. The size of the resulting image varies according to the distance between the opposite side of the box (film) and the center of projection. The model can further be simplified by placing the virtual film in front of the center of projection. This way the resulting image is generated in the right orientation.

One drawback of the perspective projection is that it still requires some amount of geometric calculation in order to determine the distance from a given surface point to the eye. But this computational overhead can be minimized by applying a perspective transformation that not only projects points to two dimensions (x,y image plane) but also preserves meaningful distance information (z component), which can be used to determine the visible surfaces. Such a transformation exists and is widely used in digital image synthesis. The perspective view frustum can be transformed into an orthographic frustum by moving the center of projection to infinity, so that all lines to the projection point become parallel. This is illustrated



(a) Perspective View Frustum (b) Canonical View Volume

Figure 2.1: Perspective transformation.

in Figure 2.1. The perspective view frustum with the center of projection at the origin (left) is transformed to a unit sized cube, the so called *canonical view volume* (right).

As can be seen in this figure, the view frustum is not only bound by the side planes (top, bottom, left, right) but also along the viewing direction itself. It is assumed that the view frustum starts at the *near* clipping plane and ends at the *far* clipping plane. These clipping planes are essential for an actual implementation since we cannot represent a unbounded numerical range of z.

Using 4D homogeneous coordinates, as explained in [Foley96], the perspective transformation can be written as a 4×4 matrix that is then be applied as the final transformation in a scene structure.

In the OpenGL graphics API [Woo99] the perspective transformation matrix transforms the viewing frustum into a unit cube. The 4×4 is

$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad . \tag{2.1}$$

The parameters here correspond to the rectangle that defines the front of the view frustum (r, b, n) and (l, t, n), which will be mapped to $(-1, -1, -1)$ and $(1, 1, -1)$, and the far distance value f, which will be mapped to a z value of 1. Other graphics API, e.g. DirectX [Microsoft00], perform the mapping to a half-cube with z values between 0 and 1.

2.1.2 Z-Buffer

A very simple hidden surface removal technique is the *z-buffer* algorithm introduced by [Catmull75]. This brute-force, image-spaced approach requires an image buffer that is not only capable of storing a color value for each pixel, but also the corresponding distance to the viewer (depth). For each primitive being rasterized, color values are only updated if the pixel's actual distance is less than the value stored in the buffer, otherwise the pixel is rejected. For pixels passing the test, the stored depth and color are replaced by the new values. Using the canonical view frustum introduced before, the normalized z coordinate can be used as an appropriate distance, instead of computing the euclidean distance. It can easily be seen that the complexity of this approach is linear, since it only depends on the number of pixels in the output image and the number of primitives being rasterized.

During the 1970s, this approach was not very popular and only used in specific cases, e.g. for curved surfaces where polygonal sorting was too complicated. For general polygonal scenes, object-space techniques combined with hierarchical representations were more efficient for that specific task. One main argument against the z-buffer was the additional memory needed for the framebuffer. In order to obtain reasonable results, it should have a resolution of 16 to 32 bits per pixel.

The popularity of the z-buffer changed immediately in the beginning of 1980, when companies like Silicon Graphics introduced special graphic computers that had dedicated z-buffer hardware, including a framebuffer capable of storing depth and color. For a hardware implementation, the z-buffer algorithm was the optimal choice. First, the algorithm itself is very simple and needs only trivial load, compare, and store operations. Second, the algorithm fits perfectly in the pipeline design since it only depends on the values stored in the framebuffer and the incoming pixel.

2.1.3 Painter's Algorithm

One of the simplest algorithm in the class of object-space algorithms is the *depth-sort* or *painter's algorithm* [Newell72]. The algorithm works in three steps:

1. Sort all polygons according to their minimum z value.

2. Check for any overlaps in z and split polygons if their z extends overlap.

3. Draw all polygons into the framebuffer from back to front.

The second step is by far the most expensive one, since it generates new geometry if polygons need to be splitted. In contrast to the z-buffer approach, the resulting priority-list does not depend on the final image resolution, and can therefore be re-used if the scene and viewing parameters remain constant. The naive implementation of Newell's approach has quadratic complexity, since every polygon has to be checked against all other polygons in the scene.

A special application for which list-priority algorithms are also used in hardware-accelerated rendering is the rendering of transparent objects. In order to correctly blend all polygons, the order of drawing needs to be back-to-front, which is not possible with the z-buffer method.

2.2 Lighting and Shading

The appearance of an object is not only dominated by its shape and position in the environment, but also by its color. Several factors influence the surface appearance of an object: The material it is made of as well as the type of illumination under which the object is viewed. In this section we will introduce the concepts of lighting and shading as it is done in digital image synthesis. We will take a close look on techniques used in hardware-accelerated rendering, in which very simplified models are common due to performance issues and hardware capabilities.

2.2.1 Radiometry

Digital image synthesis is strongly related to the physics describing the transport of light in space. We therefore introduce some of basic radiometric terms and quantities needed for the accurate description of light and shading models.

Radiant energy Q $[J]$

Planck showed that each photon carries a discrete amount of energy which is proportional to its wavelength. A photon's radiant energy is $Q = hv$, where h is Planck's constant and v the frequency of radiation. The total radiant energy is the contribution of all photons over all wavelengths.

Radiant flux Φ $[W]$

Radiant flux is the radiant energy per time $\Phi = dQ/dt$, or $\Phi = Q/t$ if energy is constant during time.

Radiant intensity I $[W/sr]$

Radiant intensity is the radiant flux per unit solid angle $I = d\Phi/d\omega$.

Irradiance E $\left[W/m^2\right]$

Irradiance is a special case of radiant intensity that describes the radiant energy per unit area incident onto a differential surface point x, $E(x) = d\Phi/dA$.

Radiosity B $\left[W/m^2\right]$

Radiosity is another special case of radiant energy per unit area referring to energy leaving a surface.

Radiance L $\left[W/m^2 sr\right]$

Radiance is the radiant flux per unit projected area per unit solid angle (incident or outgoing).

$$L(x,\omega) = \frac{d^2\Phi}{\cos\theta dA d\omega} \tag{2.2}$$

For shading computation, radiance is on of the most important quantities since it describes how many photons per time arrive at a differential area on a surface from a specific direction.

2.2.2 Reflectance Models

Defining the optical properties of a material requires a formulation of how incoming light is reflected back into the environment. A common way of specifying this is the *bidirectional reflection distribution function* (BRDF)

$$f_r(x, \vec{\omega}_i \rightarrow \vec{\omega}_o) := \frac{dL_0(x, \vec{\omega}_o)}{L_i(x, \vec{\omega}_i) cos\theta_i d\omega_i} \tag{2.3}$$

that describes how much of the energy arriving at point x from a direction $\vec{\omega}_i$ is reflected towards an outgoing direction $\vec{\omega}_o$. Although the BRDF includes some simplified assumptions, e.g. neglecting effects like fluorescence, phosphorescence, or participating media, it is capable of modeling most of the important physical effects needed in digital image synthesis. A more detailed description of BRDFs including all physical properties and assumptions can be found in [Glassner95].

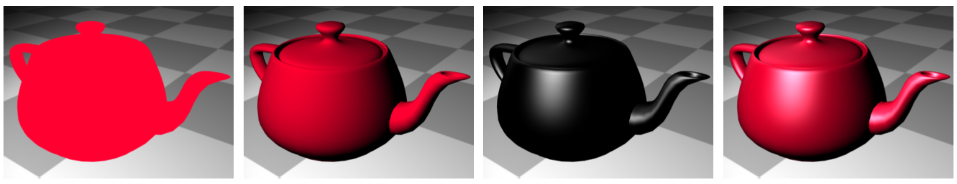
As can be seen in Equation 2.3, the BRDF in its general form is a 6-dimensional function, depending on the surface point x as well as on the incoming ($\vec{\omega}_i$) and outgoing direction ($\vec{\omega}_o$). Accurately representing a material by measuring its BRDF is often too complex due to the large amount of memory needed to store the sampled data. Many rendering systems therefore choose to use *reflection models* instead. These models describe the characteristics of specific materials using only a few parameters instead of a high-dimensional BRDF.

In the following we describe some of the reflection models that are commonly used in hardware-accelerated rendering. Implementing a reflection model in hardware requires that the model itself can be evaluated very efficiently but is still general enough to represent a wide range of materials.

The simplest type of reflection is *ambient reflection*. Here it is assumed that energy (light) is arriving uniformly from all directions. This results in a constant color across the surface, depending only on the fraction of incoming light that is reflected. Ambient reflection is used e.g. in CAD/CAM applications, where the realistic shading of solids is not required, or as an approximation for the indirect illumination in a scene.

Using *diffuse* or *Lambertian reflection* the outgoing energy is equal for all directions but depends on the incoming direction. Viewing the object from different directions has no effect on the appearance, but moving the object or light source may change the appearance. The physical motivation for diffuse reflection is light that is scattered multiple times inside the material and leaves the surface with no preferred direction.

A simple reflection model that adds glossy effects, suitable for shiny plastic surfaces, was proposed by [Bui-Tuong75], known as the Phong model. Trading in physical correctness for efficiency, the model uses a power of the cosine between viewing and reflection direction to approximate glossy reflections. [Blinn77] presented a modified version of Phong's original model which uses the cosine between the surface normal and the halfway vector (vector between light and viewing direction). This model is widely known as the *Blinn-Phong* reflection model. Many graphics architectures support a variant of the Phong or Blinn-Phong model in hardware, often extended by some more parameters to give the user better control over the surface appearance. With graphics hardware becoming more and more powerful and flexible, more complex, realistic reflection models can be used for hardware-accelerated rendering. A detailed description of such techniques can be found in [Kautz03] and [Heidrich99].



(a) ambient (b) diffuse (c) specular (d) diff.+spec.

Figure 2.2: Examples of reflection models.

To summarize our short tour of reflection models, Figure 2.2 should give an impression of ambient, diffuse, specular and a combination of diffuse and specular reflection.

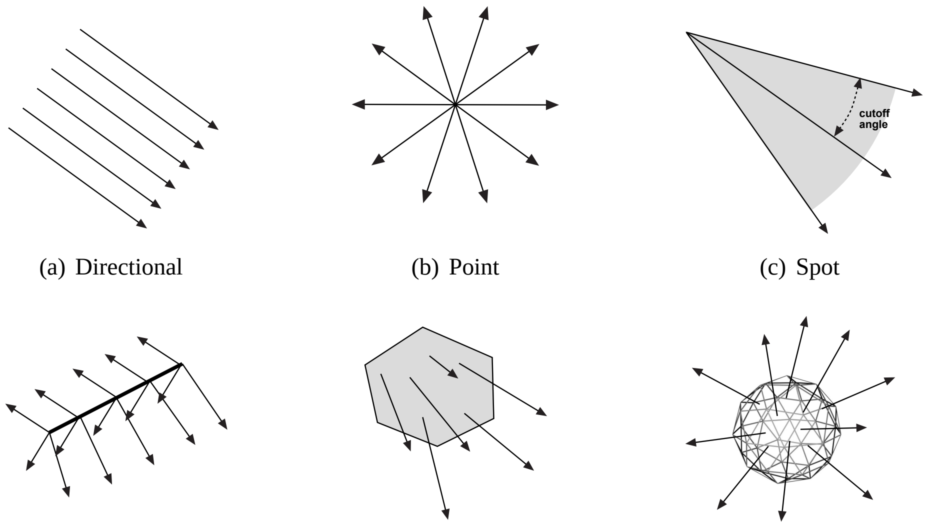
2.2.3 Light Sources

In the previous part we concentrated on how incoming light is reflected back to the environment. Since light does not come from nowhere, we also have to specify the sources of light. In general, a light source is an instance where parts of the incoming energy, e.g. in the form of electricity or heat, are propagated to the environment in the form of photons.

Light sources are an important component of each rendering system since they are responsible for the initial energy transported within the synthetic environment. Although realistic image synthesis would require physically correct models of real-world light source, very simplified models are common to reduce computation time.

One simple type of light source is a *directional light*, as depicted in Figure 2.3(a). Here it is assumed that light is originating from a point at infinity, so that all light rays hit the surface at the same direction. The light source is therefore defined by its direction and intensity. Directional light sources are especially useful to model sun light, for which the assumption of a far away light source holds.

To model a light source in near distance, a *point light* can be used that radiates energy from a given position equally to all directions (Figure 2.3(b)). This model can further be improved by adding additional parameters such as a main direction and a maximum angle (cut-off) that together define a cone to which illumination is restricted, as shown in Figure 2.3(c). This is known as the *spot light* model, inspired by the type of light sources used for stage lighting in theaters. To emphasize the



(a) Directional (b) Point (c) Spot

(d) Linear (e) Area (f) Volume

Figure 2.3: Types of light sources.

main direction, intensity distribution may also be decreased as the direction leaves the main direction, using e.g. an exponential intensity fall-off.

The next three models we want to describe are the so called *extended* light sources. In contrast to point or directional lights, these models describe more physically plausible types of illumination since energy is not originating from a infinitesimal small or far away source but from a line, surface, or volume.

Linear lights are mostly used to model the illumination produced by a long and thin lamp , which is approximately true for many neon lamps found in buildings. A simple linear light radiates energy from all points on a line segment equally into all directions as depicted in Figure 2.3(d).

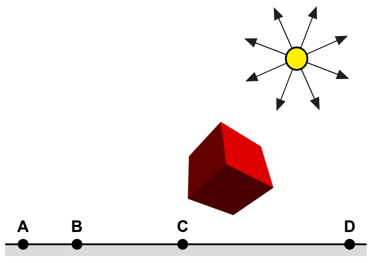
Extending linear lights with one more dimension brings us to the *area light* model (Figure 2.3(e)), where light is emitted at each point on a defined surface. The simplest area light is a triangle but in general, any surface description can be used. Area lights play an important role in realistic image synthesis since they can be used to represent real world illumination quite well. However, for hardware-accelerated rendering the illumination evaluation for this type is often too time consuming, so that a number of point or directional lights are used instead.

An even more complex type of illumination can be represented by a *volume light*. Here all points inside a volume emit light into the environment as shown in Figure 2.3(e). Due to its complex nature, volume light sources are rarely used by now, so we will neglect this specific type of illumination in this thesis.

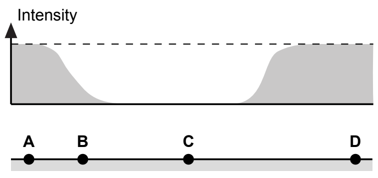
For many of the light sources introduced so far we have assumed that the intensity is uniform for all outgoing directions, except for the spot light model where intensity varies as the direction leaves the main axis. More physically correct illumination can be achieved by using a sampled representation of light source characteristics, obtained by measuring real-world luminaries. This representation is often directly available from lamp manufacturers in form of a *goniometric diagram*, from which the intensity for a given direction can be extracted. A detailed description of such industry standards can be found in [Glassner95].

2.2.4 Shadows

Rendering an object with realistic appearance we need to evaluate the reflection model on all surface points with respect to those rays of light that hit the surface at the given point. This requires an examination of the environment in which the object is placed: Firstly, the object may not be in the view of the light source, maybe completely outside the cone of a spot light. Secondly, there could be other objects in the scene that block some or all of the rays from the light source to the surface point. An example of the later situation is sketched in Figure 2.4(a). In



(a) example setup



(b) intensity curve

Figure 2.4: Example of intensity transition.

this scene we want to determine the contribution of the luminaire above at various sample points (A,B,C,D) on a given surface. At point A, all rays that are emitted in the direction to A hit the surface. For point B some of the light rays are blocked by the cube in-between. At sample C no light at all can hit the surface, whereas point D again receives the full energy.

Computing the intensity level for all sample points on the surface results in a smooth curve as shown in Figure 2.4(b). We can classify the points by either fully lit (A,D), partially lit/shadowed (B), or fully shadowed (C). It is clear that this intensity curve does not only depend on the spatial arrangement of receiver surface, blocker, and luminaire but also on the type of light source used.

Given a point light source which emits light from single point in space, there's only one ray of light that can potentially hit the surface point, which is exactly the ray originating from the light source and directed towards the surface point. The

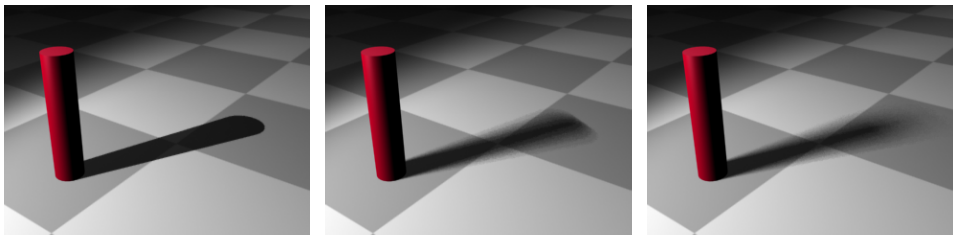
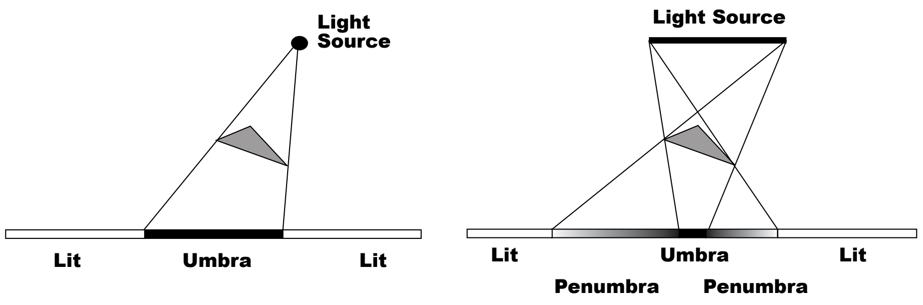

(a) point (b) linear (c) area

Figure 2.5: Shadows from different types of light sources.

corresponding intensity level can therefore only be 100% (fully lit) or 0% (fully shadowed), depending whether the ray is blocked or not. The resulting shadow will exhibit sharp boundaries where the intensity level changes as can be seen in the example scene shown in Figure 2.5(a). We will refer to this type of shadow as *hard shadow* or *umbra*. Sharp shadow boundaries are common to all non-extended light sources, such as directional, point, and spot lights.

Using a linear light source, a number of light rays may hit a specific surface point, all originating from sample points on the line segment associated with the light source. Contribution therefore varies between 0% and 100%, resulting in a smooth shadow transition as shown in Figure 2.5(b). Linear light sources have a specific characteristic that can be observed at the top of the cast shadow in Figure 2.5(b): Smooth shadow regions are only generated for blocker edges that are not parallel to light source's line segment. In the parallel case a hard shadow transition will be visible, since all rays are blocked when crossing the given edge. We refer to the region in which the transition from lit to shadowed takes place as the *penumbra*. A shadow consisting of penumbra and umbra regions is called *soft shadow*.



(a) point (b) linear

Figure 2.6: Hard vs. soft shadows.

For a linear light source we can easily visualize the geometric relationship between blocker, receiver, and light source and construct the shadow boundaries. Figure 2.6 shows such a 2D arrangement. On the left side, a point light source is used. As we can see, the contribution of the light source changes at two points on the receiver, resulting in a lit-umbra-lit transition. Using an extended light source, e.g. a linear one as shown on the right side of Figure 2.6, we can determine the relevant points at which illumination changes by drawing lines between the extremal points of the blocker and the end points of the linear light. This gives us the regions on the receiver in which the light source is fully visible, partially visible/occluded, and fully occluded, resulting in a lit-penumbra-umbra-penumbra-lit transition.

Overall smooth shadows can be obtained when using an area light source (Figure 2.5(c). As explained before, an area light is of finite size where every point on the surface can emit rays of light. As in the linear case, each point on a surface may be hit by all, some, or none of the light rays, resulting in a *soft shadow* if blocking objects are in between.

Let us now examine the physical relationship between light emitted and light received in more detail. According to [Agrawala00] the irradiance caused by an area light source with uniform energy distribution is given by

$$E = \int_A \frac{L \cos\theta_i \cos\theta_l}{\pi r^2} V \, dA \tag{2.4}$$

where θ_i is the angle at which the surface is hit, θ_l is the angle between ray and light source normal, and L is the light's radiance. A special role in this equation is the visibility mask V, which should be set to 1 for rays that hit the surface or 0 for rays blocked by objects in the environment. Or in other words, V is only set to 1 if the light source sample point can be seen from the actual surface point.

A common way to speed up evaluation of this integral is to neglect V in the irradiance calculation and use an average visibility

$$ATT = \frac{1}{A} \int_A V \, dA \tag{2.5}$$

that is later applied to dim the shading result [Agrawala00]. Although this approach is not physically correct, it is widely used in many rendering systems due to the following reasons. Firstly, the human eye is very sensible to low-quality shadows than it is to low-quality, non-shadowed shading. A renderer can therefore choose to use more light source samples for computing the attenuation factor and spend less work on the evaluation of the reflection model. Secondly, separate visibility gives way for algorithms that do not sample the light source or analytically compute the irradiance but output a rough, fast approximation for ATT. Such algorithms will be shown in Part III of this thesis.

2.3 Hardware-accelerated Rendering

Having introduced some aspects of realistic image synthesis, like materials and light sources, we will now focus on one specific type of rendering system, which is hardware-accelerated rendering. Using hardware customized for specific tasks, such as lighting or hidden-surface removal via the z-buffer, has an enormous impact on the rendering speed. On the other hand, specialized hardware restricts rendering to those algorithms that can be implemented using the given functionality.

Nearly all graphics boards available today are rasterization based. Rasterization is the process of projecting each primitive into 2D screen coordinates and generating individual pixels for all of its covered area. These pixels are then written to the framebuffer, which is later used to drive the output display device.

One major aspect of hardware-accelerated rendering is the programming mechanism, which is used to configure and access the graphics board. While it would be possible to directly access the hardware via low-level drivers, a more sophisticated programming model is needed in order to make application development possible. In the past, several hardware vendors tried to push their specific graphics API (application program interface) but only two were accepted by the graphics community and became industry standards. One is Microsoft's DirectX [Microsoft00], which is the most used graphics API for games since it provides a common framework that not only supports graphics hardware but also other devices, such as sound cards or input devices. One major drawback of DirectX is that it can only be used for hardware running the Microsoft Windows operating system. The second industry standard is OpenGL [Woo99, Segal98], an open platform graphics API, which evolved from SGI's Iris GL, and is available for many operating systems. Since both APIs can be used to control the same hardware device, the functionality provided is nearly identical. In this thesis we choose to use OpenGL as the underlying graphics API, since it is more widely used in academic research today. All of the algorithms can of course be implemented using DirectX.

Graphics hardware today is designed in a pipeline fashion, similar to the traditional rendering pipeline proposed by [Foley96]. Figure 2.7 depicts a rendering pipeline which is a simplified version of the model used in OpenGL. The pipeline consists of three main stages: The *application* stage, the execution environment in which the application is running, is responsible for configuring the pipeline and feeding data to the subsequent stage, which is the *geometry processing* stage. During this stage, the transformation and lighting takes place. The *rasterization* part is then responsible for scan-converting the incoming geometry and further processing the resulting fragments. At the end of the pipeline is the *framebuffer*, where incoming fragments are stored.

We will now examine the individual stages of the rendering pipeline in more detail, using the definitions and specific order of operations according to OpenGL. Since the OpenGL rendering pipeline is quite complex, we only focus on the most important parts. For a complete overview and more details see [Woo99].

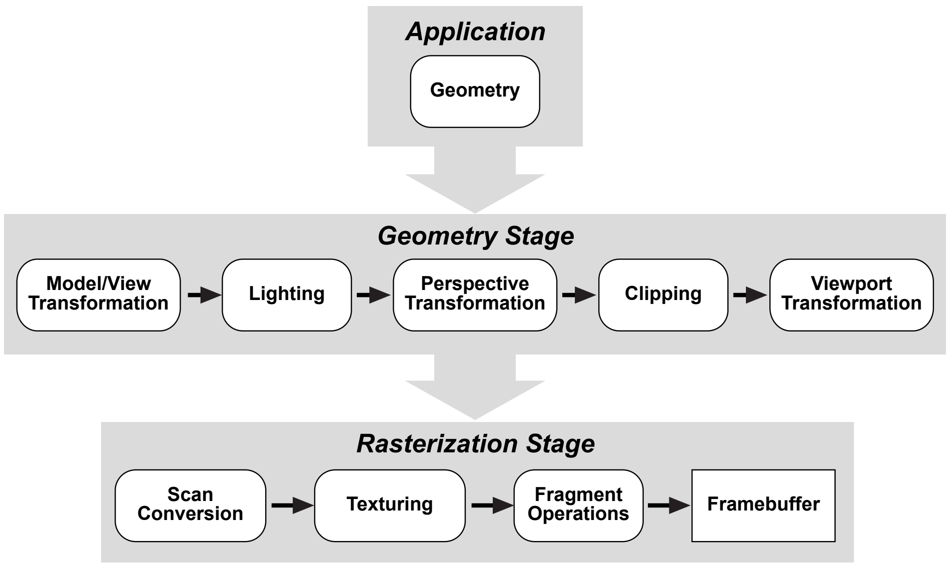


Figure 2.7: Rendering pipeline.

2.3.1 Application

The application stage is the control and data instance of the rendering pipeline. As the initial stage in the pipeline, the application is not only responsible for configuring the rendering pipeline, e.g. setting up the image resolution, downloading textures, or defining light and camera parameters, but also for feeding the geometry stage with data.

Geometric data in the context of hardware-accelerated rendering is defined in terms of primitives that are supported by the rendering architecture. OpenGL has a number of geometric primitives that a conformant implementation has to support, e.g. points, triangles, quadrilaterals, convex polygons, and connected groups of primitives (strips, fans). Primitives are either transferred as a sequence of vertex positions or using indices referring to a previously defined array of vertices.

As can be seen, the application stage has no fixed functionality and can therefore only be executed in software, where the developer has full control over the implementation.

2.3.2 Geometry Stage

Primitives arriving at the geometry stage are further processed in several sub-stages working on a per-primitive or per-vertex basis.

The first task is to transform all vertices into a global coordinate system. This transformation consists of bringing the vertices to *world space* using the actual modeling matrix. Next comes the transformation to *eye space*, where the transformation is given by the matrix incorporating the viewing parameters (position, direction, field-of-view) of the camera. Since the representation in world space coordinates is not needed in the pipeline, many architectures, such as OpenGL, use a combined *modelview transformation*, which directly transforms vertices from their local coordinate system to eye space.

In order to give objects a realistic appearance, one major task of the geometry stage is the lighting calculation. OpenGL uses for this task a modified version of the Blinn-Phong model [Blinn77] introduced in Section 2.2.2, which is evaluated per vertex. For efficiency reasons, the types of light sources OpenGL directly supports are limited to the simple, non-extended ones, namely direction, point, and spot light, that were explained in Section 2.2.3. Evaluation of the Blinn-Phong model implies that the application stage not only provides the coordinates of the vertex itself but also several other attributes and parameters. Attributes are additional properties that can vary with each vertex, such as a normal vector, color, texture coordinates, and several other properties. These attributes are specified in conjunction with the vertex position and can therefore also be transferred directly by value or indirectly using indices to attribute arrays. Parameters, on the other hand, are those values that remain constant for a number of primitives, e.g. describing the material of an object that consists of many triangles. The values for these are specified in a setup phase that configures the hardware before any geometry is passed through. While the only additional attribute for the Blinn-Phong model is the normal vector, there is quite a large number of parameters that affect this computation, e.g. several positions and directions for the light sources in the scene as well as the ambient, diffuse, and specular parameters of the material itself.

The geometry stage is also responsible for transforming the vertices into the canonical view volume according to the perspective or orthographic projection shown in Section 2.1.1. Since this transformation distorts the spatial relationship needed for lighting computation, it cannot be combined with modelview transformation. The canonical view volume simplifies the process of *clipping*. Here all primitives completely outside of the unit cube are rejected, while for primitives partially inside/outside of the view volume intersections with the side planes have to be computed in order to pass only those parts to subsequent stages that are visible.

The remaining task of this stage is the *viewport transformation*, which brings all vertices passing the clipping stage to *screen coordinates*.

Programmable Transform and Lighting

The growing demand of having more realistic reflection models than Blinn-Phong and more control over the geometric transformations has led to graphics hardware that supports programmable transform and lighting. The first user-programmable vertex engine was proposed by [Lindholm01] and realized in the GeForce-series of graphics cards by NVIDIA. Several other hardware vendors proposed similar vertex programming models, which differed in the number of operations and parameters supported. Recently, the OpenGL architectural review board (ARB) merged the various programming models into a common, vendor-independent specification that we now want to explain in more detail.

Vertex shaders, also called *vertex programs*, replace parts of the fixed-function transform and lighting operations that are computed per-vertex, such as modelview and projection matrix transformation, normal transformation, per-vertex lighting, texture coordinate generation, and many others. Operations that are not affected are clipping, perspective divide, viewport transformation, color clamping, and all other operations in subsequent parts of the geometry stage. Figure 2.8 illustration the

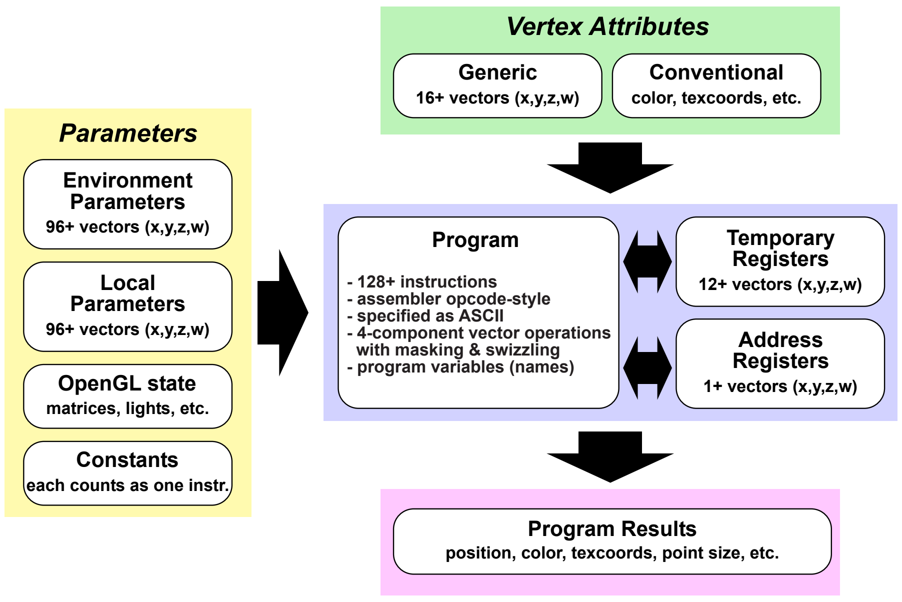


Figure 2.8: Vertex program execution environment.

environment according to the OpenGL ARB specification in which a user-defined vertex shader is executed. The central part of this environment is the vertex program, an assembler program specified as an ASCII-string, which will be executed on the graphics hardware. The instruction set for these programs is based on vectorized operations, all working on 4-component floating point vectors. Besides the standard instructions (move, add, sub, etc.) there are also a number of instructions specialized for vertex processing, e.g. to compute dot products or reciprocal square roots. A vertex program works on one vertex at a time, so there is no possibility to access or modify connectivity information or access values of adjacent vertices of the primitive.

To store intermediate values, the vertex shader has read/write access to a number of temporary registers. There is also one special register, called the address register, which can be used to perform indirect lookups from the parameter sets described later.

As in the fixed-function geometry stage, there are a number of attributes and parameters which the vertex shader takes as input. Attributes are the conventional OpenGL per-vertex attributes (position, color, texture coordinates, etc.) as well as a set of 16 or more generic attributes that can be used to pass additional data per-vertex. There are four types of parameters that can be used within a vertex program: A set of environment parameters, that are used to define global values used by a number of vertex programs. A set of local parameters, that are bound to a specific vertex shader. A number of parameters that correspond to the OpenGL state relevant for vertex processing, e.g. the actual modelview matrix or light source parameters. These parameters are automatically set by OpenGL. Constant parameters are implicitly set inside the code of the vertex shader.

After evaluation, the vertex shader outputs the transformed vertex position (in clip space) and a number of additional attributes, such as texture coordinates, color, fog factor, point size, etc., which are then passed to the subsequent clipping stage.

Programmable vertex processing is a powerful tool which enables the implementation of complex algorithms on the graphics hardware. Examples of use include all kinds of reflection models or geometric deformation, which can be seen in recent computer games.

The complete ARB style vertex programs are currently only supported on newer graphics hardware, like the ATI Radeon 9700/9800 or NVIDIA's GeForce FX series. Older cards only support some parts of the extension. In this case it is up to the driver to decide whether it can execute a given shader on the hardware or if it has to be evaluated in software.

2.3.3 Rasterization Stage

The transformed and lit vertices arriving at the rasterization stage are first processed by the *scan conversion* unit, which generates so called *fragments* for the screen space region occupied by the primitive. A fragment can be seen as an extended pixel, that not only represents a color value, but also carries additional attributes, such as a depth and alpha value and texture coordinates. Except for the texture coordinate sets, which are perspectively correct interpolated, all values are linearly interpolated using the per-vertex values.

If texturing is enabled, the fragment's texture coordinates are then used to do the texture lookups in the defined images. Texturing can be done in several ways, using one up to four dimensional textures and different filter techniques, such as nearest neighbor, bilinear, or even trilinear using mipmap textures. The application has also some control of how the texture's color is combined with the interpolated vertex color and other color values.

After all values of the fragments have been set, a number of test are applied. The *scissor test* is used to reject fragments that are outside of a specified screen-space rectangle. The *alpha test* rejects fragments depending on their alpha value by comparing it with a given reference value. The *stencil test* conditionally eliminates a fragment based on the corresponding value in the stencil buffer. The *depth test* is used for hidden-surface removal. Here the depth value of the fragment is compared with the value stored in the z-buffer, explained in Section 2.1.2.

A fragment passing all those tests is written to the *framebuffer*. There are several modes to combine the incoming color with the color already stored, which is called *blending*. Additionally *dithering*, *logical operations*, and *masking*, can be applied. See [Woo99] for more details.

The framebuffer itself is a very powerful tool. Not only can it be used to store the final image, but it can also be utilized as normal memory, e.g. for storing intermediate results. One special functionality that is now available on many graphics cards is the rendering to virtual buffers, so called *offscreen buffers*. An offscreen buffer is completely separated from the display device but can be used just like the normal framebuffer. An example application of offscreen buffers is *render-to-texture*, a process where the image generated in a previous rendering pass is later used as a texture image. Offscreen buffers may also support more bits per color channel, up to floating point precision on recent graphics cards.

Programmable Per-Fragment Computations

The inflexibility of the rasterization stage has led to a number of OpenGL extensions that added more functionality. NVIDIA proposed a simple but very powerful mechanism called *register combiners* [NVIDIA02] that replaced the texture, color sum, and fog computation. Recently, the OpenGL ARB has specified a general programming model that greatly simplifies the process of configuring those parts of the rasterization stage.

The so called *fragment shaders* (or *pixel shaders*), are small assembler programs that are evaluated in an execution environment which is similar to the vertex shaders introduced in Section 2.3.2.

A fragment shader takes as input the interpolated color values and texture coordinates generated in the geometry stage. In addition to those attributes there are also a number of parameters that are either user-defined or represent parts of the OpenGL state.

The instruction set supported is similar to the vertex shaders, but has additional operations that can be used to perform texture lookups. In contrast to the fixed-function pipeline texture lookups are no longer limited to the specified texture coordinates. A fragment shader is free to sample the texture at any position or use the result of one texture lookup as texture coordinates for a second lookup, a process also known as *dependent texturing*.

The result of a fragment program is the fragment's color and alpha value, which is then passed to the per-fragment operations. It is also possible to replace the fragment's depth value, but a fragment program cannot change the fragment's screen space position.

Currently, fragment programs do not support branching or looping, due to performance reasons, but this may change in future hardware.

2.3.4 Graphics Architectures

We will now take a closer look on some graphics architectures focusing on how the rendering pipeline is realized and what additional functionality is available. Currently, the time slot in which a specific graphics card can be seen as the state-of-the-art hardware is only about 6 months, or even less, so we will focus only on those systems that were used for implementing and testing the algorithms proposed in the following chapters.

The design of graphics hardware drastically changed during the last 5 to 10 years. Hardware-accelerated rendering was for a long time dominated by large, massive-parallel graphics computers, e.g. SGI's Onyx with the InfiniteReality graphics subsystem. With the growing market of consumer class PC-systems and video games, affordable, single-chip graphics cards became popular.

NVIDIA GeForce 256 (1999)

With the release of this card in 1999, hardware-accelerated rendering on consumer class PC-hardware set a new standard. NVIDIA presented the first single-chip architecture that did geometry processing (transform and lighting) and rasterization. This chip was able to transform, clip, and light up to 25 million triangles per second. The rasterization stage was able to perform at up to 480 million pixels per second, using four parallel pixel pipelines and up to two textures simultaneously. It was also the first card that supported a more general programming model during fragment processing with the register combiner extension. At the end of 1999, NVIDIA already presented a modified version of this card called GeForce2, which mainly offered higher performance rather than new features.

SGI Octane/VPro (2000)

With the VPro graphics boards used in the Octane workstation series, SGI presented their first single-chip OpenGL rendering engine. Although the peak geometry rate of 7 million triangles per second and a maximum fill rate of 425 million pixels per second could not compete with the peak numbers of PC-cards at the time, it had many features which are relevant for scientific and industrial applications: Full hardware support for the OpenGL imaging subset, an extension specialized to imaging operations, such as convolution, color conversion, histogram computation etc. Higher color precision using up to 12 bits per channel in the frame buffer and 16 bits during the rasterization stage.

In contrast to the PC-hardware, this system is located in the high-end unix workstation price segment ($>$ 15.000 USD). Due to its fast memory interface and accelerated 2D-imaging capabilities, it is still widely used today, e.g. for 2D compositing or medical applications where large amounts of data need to be processed.

NVIDIA GeForce3 (2001)

The flexibility of the hardware rendering pipeline further improved with the release of NVIDIA's GeForce3. Besides the higher fill and geometry rate, the card offered several new features that replaced parts of the fixed-function OpenGL pipeline. It was the first card that supported programmable vertex processing using vertex shaders, not yet at the level of the shown ARB-style shaders (Section 2.3.2), but still very flexible. Programmability was also improved during texturing using so-called *texture shaders* that could e.g. be used to perform dependent texture lookups or dot products on texture coordinates.

ATI Radeon 9700 (2002)

The next major step in graphics technology was the release of cards supporting Microsoft's DirectX 9 standard, such as ATI's Radeon 9700. This card supports the full-featured ARB-style vertex shaders. The geometry stage was further extended to process higher-order surfaces and displacement mapping. All computations during rasterization are performed at floating point precision, supporting ARB-style fragment programs (Section 2.3.3) with up to 16 different texture images and 32 individual lookups. Full floating point precision is also available for offscreen buffers and texture images.

NVIDIA GeForceFX 5800 (2003)

In 2003, the buzzword for state-of-the-art graphics cards is *cinematic rendering*, since the quality of the rendered images is getting close to software rendering.

This card increased the flexibility of the rendering pipeline by supporting complex vertex shaders (over 65000 instructions), with flow control (looping, branching, subroutines), as well as more complex fragment shaders with full floating point precision.

Shading Languages

With respect to software developers, the functionality available on those type of cards is also problematic. In order to take full advantage of the features, an application has to be customized to run on a specific card. Besides the standard OpenGL functionality, recent graphics boards may offer over 70 extensions to OpenGL, where a large number of extensions are proprietary. It is therefore necessary to define a more general programming model for flexible graphics pipelines. A first effort in this direction was presented by [Mark03], proposing a C-style shading language for vertex and fragment processing. The ongoing specification of OpenGL 2.0 will even go further since it will not only include a powerful shading language [Kessenich03] but also a very general memory model that simplifies the management of framebuffers, texture images, vertex arrays, etc.

Chapter 3

Overview of Shadow Techniques

Since shadows are such an important visual effect, it is not surprising that a host of literature by many researchers is available on this topic. In this chapter we will only discuss some methods, focusing on those suitable for interactive and real-time applications, as well as on algorithms which are related to the methods proposed in the following chapters. For a general overview on shadow techniques, we recommend Woo's survey on shadow algorithms [Woo90] as a good starting point. An excellent overview of real-time soft shadows techniques was presented by Hasenfratz et al. [Hasenfratz03].

As already mentioned in the previous chapter, shadow algorithms are in a way similar to techniques used for hidden-surface removal. To determine whether a given surface is lit or in shadow, we have to check if the surface is visible from the light source. Using a hidden-surface algorithm we therefore place a virtual camera at the light source's position and resolve visibility for all objects in the scene. The result of this step can then be used to selectively activate/deactivate the light source's contribution during shading calculation. Many shadow techniques are therefore adapted versions of known hidden-surface techniques.

We will now review a number of prominent shadow techniques, discussing their pros and cons, and showing how they can (or why they can not) be implemented on graphics hardware.

3.1 Classification of Shadow Techniques

3.1.1 Object-space vs. Image-space Shadow Computation

In the field of hardware accelerated, interactive rendering, shadow algorithms are mainly categorized by the space in which the calculation takes place. One class are algorithms that compute shadow information in *object-space*, using the scene's geometry as input data. Object-space techniques in general produce very accurate, precise shadows since most of the available information is used. The drawback of these methods is computational complexity: A naive, straight-forward object-space shadow algorithm has quadratic complexity in the number of objects since every pair of objects has to be checked for a shadow situation. For large, complex environments the processing time can therefore be enormous since every calculation will be performed at full floating point precision.

The contrast of object-space shadow techniques are algorithms that work on sampled representations of the scene. These *image-space* shadow algorithms operate on one or more render images, e.g. depth images. In general, image-space techniques are well suited for hardware-accelerated rendering since the hardware is optimized for rendering images and performing operations on images, like texture mapping. A major drawback of computing shadows in image-space is due to sampling artifacts. Shadow quality may suffer from low-sampled scene representations or numerical problems when performing operations on these images.

There is also a third class of methods often used in computer games. Instead of computing the shadow by analyzing the scene, the shadow region is sketched using additional geometry or texture maps. For example the shadow of a character on a planar surface can be approximated by a simple disc like shape that moves and scales according to the character's movement. Although these fake methods are very common, we will not discuss them here and focus on those methods that compute *real* shadows.

3.1.2 Hard vs. Soft Shadows

Another important aspect are the types of light sources that a specific shadow method can handle. As seen in Section 2.2.3 and 2.2.4, extended light sources cause very complex shadow transitions, whereas point or directional light sources exhibit only sharp shadow boundaries. In order to efficiently compute shadow regions, many algorithms therefore focus on specific types of light sources. In the context of hardware-accelerated rendering shadow computation for non-extended light sources is possible at real-time frame rates, even for very complex scenes, while extended light sources are still rarely used due to the high computational overhead.

The classification of hard vs. soft shadow techniques is not as clear as e.g. the object- and image-space classification. In theory, any soft shadow technique may also be used to produce hard shadows simply by scaling the extend of the light source down so that the result cannot be distinguished from a single light source sample. Since shadow computation for extended light sources is very complicated, this is of course no common technique.

What is more interesting is the reverse approach: Given a hard shadow algorithm, we can adopt it to produce soft shadows by approximating the extended light source by a number of point samples. In Section 2.2.4 we showed how visibility can be separated from shading using the attenuation factor (Equation 2.5). Instead of analytically integrating visibility V over the light source's area, we now accumulate a number of visibilities V_i taken from discreet sample points on the light source:

$$ATT = \frac{1}{A}\int_A V\,dA \qquad \approx \qquad ATT' = \frac{1}{N}\sum_i^N V_i \tag{3.1}$$

Using a sufficient large number of samples N, the attenuation factor ATT' should be a good approximation for ATT. This sampling approach, however, suffers from quantization artifacts. In other words, with N light source samples we can only discriminate $N-1$ levels of penumbra in addition to the umbra and the completely lit regions. This can be seen in the example scene in Figure 3.1. Here we evaluated

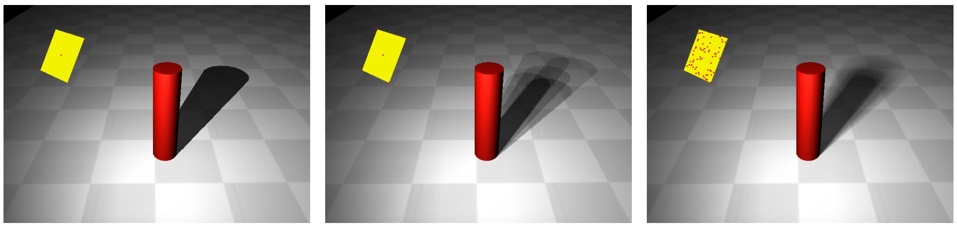

(a) 1 sample (b) 5 samples (c) 64 samples

Figure 3.1: Sampling the light source.

ATT' using a hard shadow algorithm that was applied for one sample ($N = 1$), five samples ($N = 5$), and 64 samples ($N = 64$). For $N = 1$, the result is a hard shadow transition, since only one boolean shadow value per pixel is available. For $N = 5$, the discreet sampling is clearly visible. Using $N = 64$, we can generate 63 levels of penumbra. Quantization artifacts are still visible, but for an interactive application the shadow quality may be sufficient. However, for complex environments the hard shadow technique needs to be very efficient in order to achieve reasonable frame rates.

3.1.3 Hardware-accelerated vs. Software-based Methods

The classification of hardware-accelerated or pure software methods is another important aspect, especially in this thesis. The pipeline architecture shown in the previous chapter has a direct influence on the design and implementation of a hardware-accelerated shadow technique. A software-based method can take advantage of all computational power and flexibility that CPU and memory subsystem provide, e.g. random memory accesses, subroutine calls, or complex numerical computations.

In contrast to this, graphics hardware has a very limited set of resources. The available data that can be used as input or output is more or less fixed and also the number of operations is, compared to software implementations, very limited.

This makes the design of efficient, hardware-accelerated shadow techniques a very difficult task. Graphics cards are optimized for *local* processing, meaning that all computations, e.g. lighting, access only a small subset of the scene description. Since shadows are a *global* effect, for which the spatial arrangement of objects and lights in the scene is important, an algorithm has to split the input data into smaller sets that can be handled by the hardware. This allows the computation on a per-primitive, per-vertex, or per-fragment basis, taking advantage of the hardware's capabilities.

3.2 A Review of Shadow Techniques

3.2.1 Shadow Rays

In ray-tracing [Whitted80], a shadow ray is cast towards the light source to obtain an attenuation factor for point and directional light sources. In a brute-force implementation each shadow ray has to be tested for intersection with all objects in the scene. On newer graphics hardware, which support programmable vertex and fragment stages, the shadow ray approach can be realized by storing the scene description as textures and computing intersections using fragment shader, as e.g. shown by [Purcell02]. Although this hardware-accelerated ray tracing implementation runs at interactive frame rates for scenes of moderate complexity, it does not perform well for dynamic environments. For interactive applications, pure software based ray tracing on a PC-cluster [Wald01] or with dedicated ray tracing hardware [Schmittler02], is an alternative to hardware-accelerated rendering. The shadow ray method can also be extended to distribution ray-tracing [Cook84] for the rendering of soft shadows. Here the light source is sampled by a number of rays, as explained in Section 3.1.2.

3.2.2 Geometric Analysis

Some researchers use a geometrical analysis of the scene to find regions of the scene where the whole light source is visible, the whole light source is occluded (umbra), and regions where parts of the light source are visible (penumbra). These methods work in object space by either generating discontinuities on the illuminated objects (discontinuity meshing along the lines of Heckbert [Heckbert92]), or by backprojecting the scene onto the light source, as done for example by Drettakis and Fiume [Drettakis94], Ouellette and Fiume [Ouellette99b], and Stewart and Ghali [Stewart94].

After the discontinuities have been geometrically analyzed, the actual shading of each point can be performed analytically, or again using sampling. Common to both discontinuity meshing and back projection is the large geometric complexity, which makes these approaches ill suited for interactive applications or hardware implementation.

If only a small subset of the scene geometry is dynamic, e.g. in architectural scenes, the expensive re-meshing step can be optimized by analyzing and updating only the affected meshes. This way, discontinuity meshing can be used for interactive rendering, as e.g. shown by Loscos and Drettakis [Loscos97].

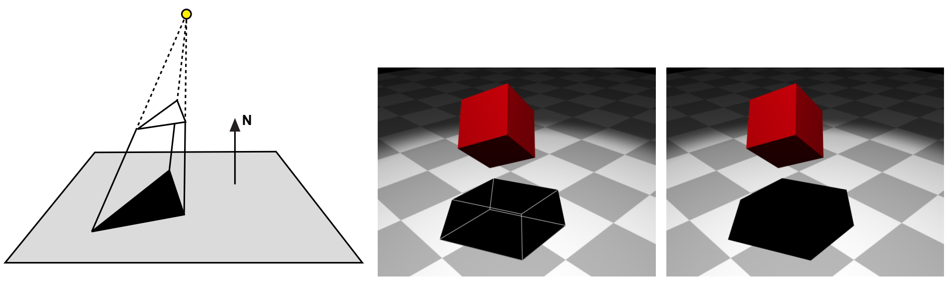
Govindaraju et al. [Govindaraju03] presented a hybrid software/hardware algorithm for interactive shadow generation in complex environments. The expensive step of geometric analysis is reduced to a small subset of the scene's geometry by using hardware-accelerated occlusion testing and hierarchical visibility culling. For regions where the analytic computation is not needed, e.g. large shadow regions, shadow maps are used (see Section 3.2.4), while exact fine detailed shadow information is covered by shadow polygons.

There has also been some work by Parker et al. [Parker98] on approximating the penumbra by manipulating the geometry of the occluders, and then assuming a point light source. This work does not yield the exact solution for the penumbra, but results in an approximation of high visual quality. We will describe a modified version of this approach in Chapter 11 using a sampled representation of the scene, which can easily be generated using graphics hardware.

3.2.3 Shadows on Planar Receivers

In the area of real-time computer graphics, especially in games, we often see special purpose algorithms that are only adequate for very specific situations, such as the projected geometry approach [Blinn88], which only works for shadows cast onto large planar objects.

Consider the setup in Figure 3.2(a). We can create the shadow on the receiver plane by drawing the triangle a second time, with the vertices projected onto the receiver and the point light as the center of projection. Using framebuffer blending we can use this additional geometry to dim the shading of the underlying plane.



(a) example setup (b) individual faces (c) resulting shadow

Figure 3.2: Projected geometry.

Figure 3.2(b) shows the individual faces of a cube projected onto the ground plane. Without the edge highlighting, as in Figure 3.2(b), the shadow looks just right.

The projection itself can be expressed as a 4×4 transformation matrix[1], which

[1]A detailed derivation of this matrix can be found in [Akenine-Möller02b].

allows us to combine it with the object's transformation matrix. For each receiver plane, we therefore have to render all objects a second time, which gives us the flattened shadow geometry.

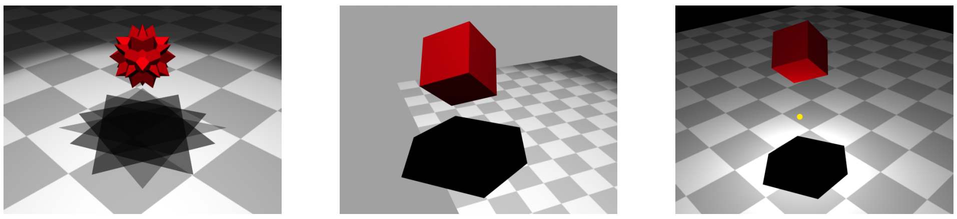

(a) multiple blending (b) surface bounds (c) anti shadow

Figure 3.3: Problems with projected geometry.

There are a number of special situations that have to be addressed. Due to limited numerical precision in the z-buffer an artifact known as *z-fighting* will appear, because it is not guaranteed that all fragments of the shadow geometry pass the depth test. One should therefore slightly offset the shadow geometry towards the light source, so that the shadow geometry is drawn above the receiver surface.

Another problematic situation is shown in Figure 3.3(a). To account for ambient illumination in the scene, it is desirable to just dim the color of the receiver plane in the shadow region, rather than drawing it fully black. If we just set up the hardware such that every shadow pixel will modulate the result in the framebuffer the shadow region will not have a consistent shading. This is due to the fact that shadow polygons will overlap when we project them onto the ground plane, resulting in multiple blending. Using the stencil buffer we can take care that only one fragment passes through, so double blending will not occur.

Up to now we assumed that the receiver is a plane, rather than a polygon with a finite size. This can lead to problems as shown in Figure 3.3(b). Here the shadow region does exceed the surface bounds of the receiver, resulting in an outside shadow. This artifact can also be easily solved using the stencil buffer. In a first pass we render the scene without the receiver polygon. Next, the receiver polygon is drawn with an additional stencil operation that set the stencil value to 1 for all fragments passing the depth test. When rendering shadow geometry, the stencil test is then used to accept only those fragments for which the stencil value is set to 1. This way drawing of shadow geometry is restricted to the receiver polygon. This approach can also be used to avoid the offset towards the light source, discussed before. If the stencil bits of the receiver surface are set, we do not need the depth test anymore, which solves the problem of z-fighting.

Using a general projection with the light source as the center of projection may result in so called *anti shadows* as shown in Figure 3.3(c). Here the cube's polygons are projected onto the plane, although the light source is between receiver and occluder, a situation where no shadow should appear. A simple solution to this would be to check for this situation and only draw those objects that are between the light and the receiver plane. Another way of avoiding anti-shadows is the use of a projective transformation that preserves z values, as shown in Section 2.1.1. With a 3D-to-3D projection we can use an additional clipping plane that rejects all geometry behind the light source.

An artifact similar to anti shadows are *false shadows*. Here geometry from below the receiver plane is projected onto the surface. This can also be solved using a clipping plane placed at the receiver plane.

If the light source and scene geometry remains constant over a number of frames, a common way is to precompute shadow textures rather than projecting the geometry in each frame. This is achieved by setting up a frustum with the light source as the center of projection and the receiver plane as the projection plane. By rendering the occluder geometry a shadow texture is generated which is then applied as a normal texture map for the receiver plane. Such precomputed shadow texture are also called *shadow masks*.

Soft Shadows on Planar Receivers

Heckbert and Herf [Heckbert97] adapted the projected geometry method for rendering soft shadows caused by area light sources on graphics hardware. This works by first choosing a sample on the area light and rendering the receiver surface with lighting from the specific sample. Next, the flattened occluder geometry is draw in black, just as in the previous algorithm. By repeating this for a number of sample points and adding up the individual images using the *accumulation buffer* [Haeberli90], a soft shadow texture is generated that will later be used as the receiver's texture map. This approach will suffer from sampling artifacts if the number of light source samples is not sufficiently high (as described in Section 3.1.2). On the other hand, the number of samples is directly proportional to the number of rendering passes and accumulation steps, which makes the method ill-suited for soft shadows in complex, dynamic environments.

An approximative approach to soft shadowing was presented by Soler and Sillion [Soler98] using convolution of blocker images. Similar to Heckbert and Herf, this method computes soft shadow textures for planar receivers, but it has the advantage that no expensive sampling of the light source is required. The key idea here is that a soft shadow can be approximated by convolution of the blocker image and an image of the area light. On modern hardware this method can utilize specialized DSP features to convolve images, leading to interactive rendering times. The main drawback of the method is the clustering of geometry, as the number of clusters is directly related to the amount of texture memory and convolution operations.

Haines [Haines01] proposed a method for approximating soft shadows by first generating a hard shadow image on the receiver plane and then compute penumbra regions using distance information obtained from the occluder's silhouette edges. The hard shadow region is generated by the projected geometry method describe previously. The penumbra regions are added by additional geometry at the hard shadow's boundaries, drawn with a gradient from black to white to approximate a smooth shadow transition. These gradient areas are shown in Figure 3.4(a): Each vertex of the occluder will cast a circular shadow area, whereas silhouette edges will cast a quadrilateral shape. The radius of each circle is varied according to the distance of the occluder vertex to the ground plane, whereas the size of the quadrilateral is determined by the circle sizes at its end points. The resulting soft

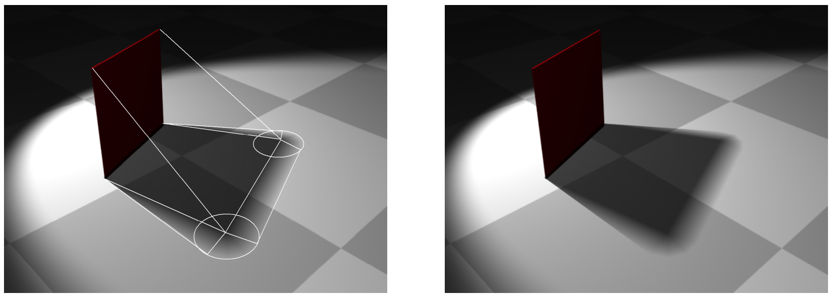

(a) penumbra geometry (b) generated shadow

Figure 3.4: Haines' soft shadow method.

shadow is depicted in Figure 3.4(b). Haines showed that the gradient overlap is not a problem when drawing threedimensional shapes (cones and sheets) and using the z-buffer to resolve visibility. This soft shadow technique is easy to implemented and well suited for hardware-accelerated rendering. A major drawback is that penumbrae are added to the hard shadow region, which makes the soft shadow look too large in some situations.

A fast soft shadow method, especially suited for technical illustrations, was proposed by Gooch et al. [Gooch99]. Here the authors project the same shadow mask multiple times onto a series of stacked planes and translate and accumulate the results onto the receiver plane.

3.2.4 Shadow Maps

Williams' shadow map algorithm [Williams78] is the fundamental idea of most shadow methods working on sampled representations of the scene.

In a first step, the scene is rendered as seen by the light source. Using the z-buffer we obtain the depth values of the frontmost pixels which are then stored away in the so called *shadow map*. In the second step the scene is rendered once again, this time from the camera's point of view. To check whether a given pixel is in shadow we transform the pixel's coordinates to the light source's coordinate system. By comparing the resulting distance value with the corresponding value stored in the shadow map we can check if a pixel is in shadow or lit by the light source. This comparison step is illustrated in Figure 3.5. Figure 3.6(a) shows an

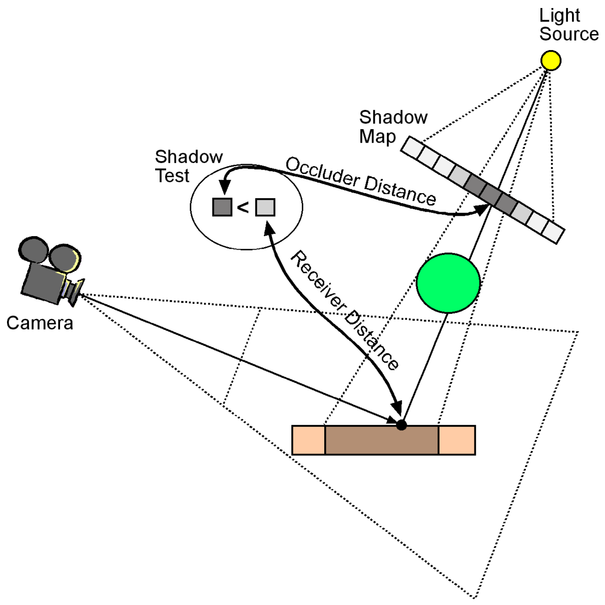


Figure 3.5: Shadow mapping illustrated.

example scene in which shadows were generated using the shadow map in Figure 3.6(b).

A hardware implementation of this algorithm using specialized hardware functionality was proposed by Segal et al. [Segal92]. Using automatic texture coordinate generation it is possible to have homogeneous texture coordinates (s,t,r,q) that are derived from the eye-space coordinate system. In conjunction with a texture matrix (which is a 4×4 matrix applied to these texture coordinates) one can implement the necessary transformation to the light source coordinate system including the perspective projection.

In order to perform the shadow test, a dedicated texture mapping mode is needed that compares the entry at $(s/q,t/q)$ in the shadow map with the computed depth value (r/q). The result of this operation is coded as color $(0,0,0,0)$ or color $(1,1,1,1)$. In addition to this shadow test mode, the graphics hardware has also to support a way of storing depth values as textures.

In OpenGL all of these capabilities are supported by two extensions: The *depth_texture* extension introduces new internal texture formats for 16, 24, or 32 bit depth values. The *shadow* extension defines two operations ($\leq$ and $\geq$) that compare (r/q) with the value stored at $(s/q,t/q)$. For a long time these extensions were only supported on high-end graphics workstations. Today hardware-support for shadow mapping is also available on consumer class graphics cards, e.g. NVIDIA's GeForce3 or ATI's Radeon.

Williams' approach is often called *backward shadow mapping*, in contrast to *forward shadow mapping* as proposed by Zhang [Zhang98]. Instead of transform-

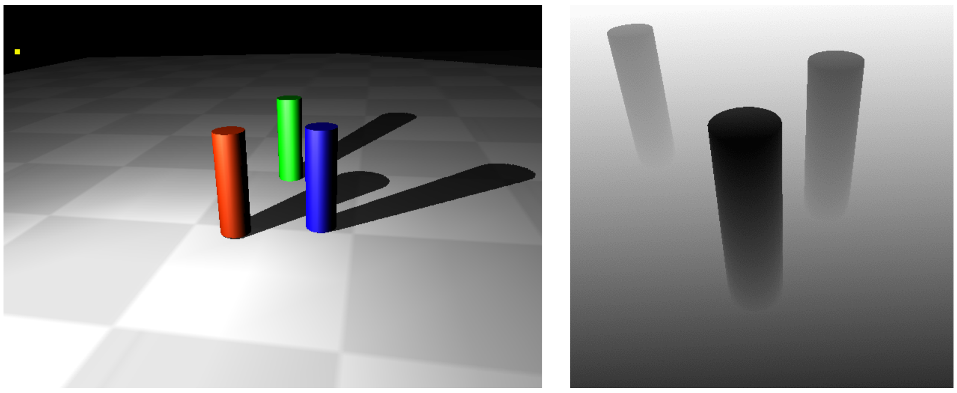

(a) final view (b) shadow map

Figure 3.6: Shadow mapping.

ing eye space pixels to the light source's coordinate system (backward), the shadow map itself is warped to eye space, resulting in a modulation image. Forward shadow mapping allows easy integration of light maps and anti-aliasing (blurring the modulation image), but the warping itself may introduce artifacts.

Color-coded Shadow Maps

The previous implementation relies on dedicated shadow mapping support, which is only available on some graphics cards and therefore not part of the standard OpenGL specification. An implementation that runs on any OpenGL hardware was presented by Heidrich [Heidrich99]. Instead of using the z-buffer a 1D ramp texture is used that converts depth values to color values. The main idea here is to use one of the color channels, e.g. alpha, for storing depth values.

The first step of the shadow map algorithm is the generation of a depth image from the light source position. Since we plan to use the alpha channel for holding the depth map, we generate this map with the following method. We render the scene with the camera located at the light position, and resolve visibility using the z-buffer. At the same time, we texture-map the whole scene with a 1D texture containing a linear ramp between 0 and 1 in the alpha channel. During the texturing step, the z coordinate in light source space is used as the texture coordinate of each vertex. This can be achieved using OpenGL's automatic texture coordinate generation. The described method yields a shadow map in the alpha channel, where the light source z is uniformly quantized between some near and some far plane containing the illuminated geometry.

Given such an alpha shadow map, we can now implement the shadow test as follows. The shadow map algorithm requires us to check for each point in space, whether its actual distance from the light source is larger than the reference distance from the shadow map. The first term, the actual distance from the light source, can be obtained again by applying the 1D alpha texture containing the linear ramp. The texture coordinates for each vertex have to be the same as during the generation of the shadow map, but this time the scene has to be rendered from the desired camera position rather than the light source position.

From this, we would like to subtract the shadow map value corresponding to each point, and check the result against 0 (a value of 0 means that the point is lit, > 0 means it is in shadow). We can achieve this subtraction by rendering the scene again, this time using the alpha shadow map as a projective texture for the whole scene, while setting up alpha blending to subtract the result from the pixel value of the previous rendering pass. Afterwards, the alpha channel contains a mask for the shadowed and the lit regions of a scene, which can be used to correctly illuminate the lit regions in a third pass. It should be noted that on a system with support for multitexturing and two simultaneous textures, the first two rendering passes can be combined into a single one. If the system even supports 3 or more simultaneous textures, all rendering passes can be combined into a single one (the generation of the shadow map still requires a separate pass), and no destination alpha channel is required in the framebuffer.

A major drawback of Heidrich's method is numerical precision due to the limited color resolution in the framebuffer, which is usually 8 bit on consumer class hardware. High-end workstations, e.g. SGI's InfiniteReality and Octane series, support up to 12 bit color component precision. Kilgard [Kilgard00] presented an extended version of Heidrich's approach which allows to use up to 16 bit depth precision by splitting and combining the high and low bytes of the 16 bit word. Recently, consumer class graphics cards support floating point offscreen buffers, but on those type of cards the shadow mapping extensions are often directly supported.

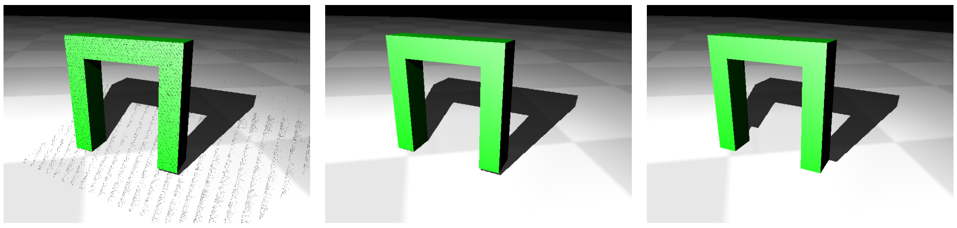
In Chapter 4 we will propose several enhancements of Heidrich's method which optimize the distribution of depth values. Chapter 5 presents a shadow map filtering scheme based on color-coded shadow maps.

Shadow Map Bias

Williams' original work suffered from sampling artifacts during the generation of the shadow map as well as when performing the shadow test.

One of the typical artifacts of a shadow map algorithm is that, due to quantization artifacts, a surface may shadow itself from the light source. This artifact has sometimes also been called *surface acne*, and leads to black spots in the middle of a lit surface, as shown in Figure 3.7(a). One solution, described by Reeves et al. [Reeves87], is to use an offset from a global interval for performing the depth comparison. This, however, can introduce other artifacts, like missing shadows in the image in Figure 3.7(c). Finding the right offset (as in Figure 3.7(b)) is therefore up to the developer.

A better solution proposed by Woo [Woo92] is to store the average depth of the first and the second intersection at each pixel of the shadow map. This way, the depth values of lit pixels are usually much smaller than the reference values in the



(a) bias too low (b) just right (c) bias too high

Figure 3.7: Shadow test bias.

depth map, and the numerical instabilities vanish. In Chapter 4 we will present a hardware-implementation of Woo's idea using color coded shadow maps.

Shadow Map Filtering

Reeves et al. [Reeves87] also proposed a filtering method called percentage closer filtering (PCF) which reduced the problem of blocky shadow edges due to low resolution shadow maps and generates smooth, anti-aliased shadow edges. Basically, PCF works by reversing the order of filtering and comparing. Instead of first filtering the texture image over some specific region and using the resulting value for further processing, PCF performs the comparison step first.

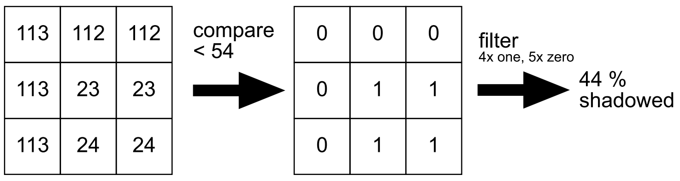


Figure 3.8: PCF illustrated.

Figure 3.8 illustrates the scheme in the case of a 3×3 region in the shadow map. Nine *z* values are compared against a given surface *z* value which results in a 3×3 binary mask from which the percentage shadowing can be calculated by simple bit counting. The region that is sampled can be determined by projecting the pixel boundary rectangle onto the shadow map. Figure 3.9 shows an example scene rendered without (left) and with (right) shadow map filtering applied. As can be seen in the close-up views, blocky shadow edges are reduced.

Hardware-support for shadow map filtering is supported on some of the newer PC-graphics cards. In Chapter 5 we will present a special PCF method that can be implemented on standard OpenGL hardware.

Reeves' approach is also often used to approximate penumbra regions by varying the filter kernel with respect to the projected footprint. This is requires a very

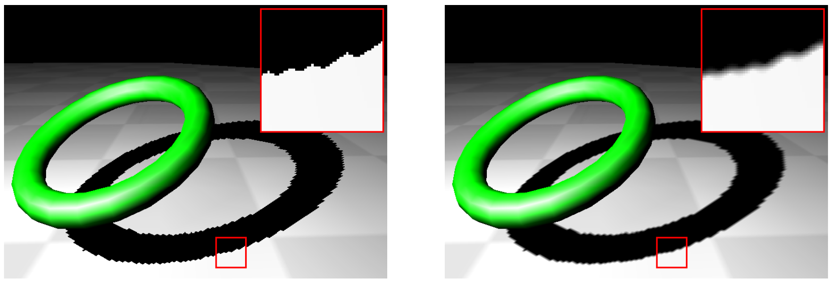

(a) no filtering (b) PCF

Figure 3.9: PCF filtering.

high resolution depth map in order to obtain soft shadows with reasonable quality.

Hourcade and Nicolas [Hourcade85] also addressed the shadow map sampling problems and came up with a method using object identifiers (priority information) and prefiltering.

A special type of shadow maps, called deep shadow maps, were presented by Lokovic and Veach [Lokovic00]. Instead of storing only a single depth value in each shadow map entry, a list of fractional visibility at each possible depth is stored. Deep shadow maps can therefore be used to capture fine detailed geometry, such as hair or fur, or even shadows from particles. A hardware-based implementation of this approach was presented by Kim and Neumann [Kim01], based on a volume slicing scheme.

Shadow Map Parameterization

When generating the shadow map in the first pass, it is also desirable to adopt the sampling rate of the shadow map for the current camera view. Recently, two papers addressed this issue.

Fernando et al. [Fernando01] came up with a method called *Adaptive Shadow Maps* (ASMs) where they presented a hierarchical refinement structure that adaptively generates shadow maps based on the camera view. ASMs can be used for hardware-accelerated rendering but require many rendering passes in order to refine the shadow map.

Stamminger et al. [Stamminger02] showed that it is also possible to compute shadow maps in the post-perspective space of the current camera view. These *Perspective Shadow Maps* (PSMs) can be directly implemented in hardware and greatly reduce shadow map aliasing. Drawbacks of the PSMs are that the shadow quality varies strongly with the setting of the camera's near and far plane and that special cases have to be handled if e.g. the light source is located behind the viewer.

In Chapter 4 we will show how a tight fitting frustum for shadow mapping can be generated automatically, which results in a better sampling rate.

The shadow map's parameterization is also important for the type of light source being used. Using Williams' original method, only spot lights with a limited cut-off angle are supported due to the perspective projection involved. In Chapter 6 we will propose a different shadow map parameterization, based on dual-paraboloid mapping [Heidrich98b], which can be used for hemi- and omnispherical light sources.

Soft Shadows using Shadow Maps

A common way to render soft shadows using shadow mapping is to sample the light source and accumulate the individual hard shadow images. With dedicated hardware-support and scenes with medium complexity this approach is practical for a small number of light source samples.

Isar et al. [Isard02] presented an implementation of this brute-force soft shadow method that distributes shadow map generation over a number of graphics PCs. This way, interactive rendering of high-quality soft shadows is possible, since each node only has to render a small number of shadow maps.

Chen and Williams [Chen93] computed soft shadows caused by area lights using image interpolation techniques. Given a small number of shadow maps from individual sample points new shadow maps are computed by interpolating between neighboring maps.

Agrawala et al. [Agrawala00] efficiently adopted image-based methods to compute soft shadows. Although their coherence-based ray tracing method does not perform at interactive rates, they also presented an approach using layered attenuation maps, which can be used in interactive applications.

Keating and Max [Keating99] used multi-layered depth images (MDIs) to approximate penumbra regions. MDIs are obtained from only a single light source sample, but store depths at multiple distances from the light source. By warping each camera pixel to the MDI frame, soft shadows are generated by depth-dependent filtering. Since traversal of the MDIs is done using ray tracing, the method is only suitable for offline rendering.

An image-space variant of Haines' [Haines01] soft shadow method for planar receiver was presented by Wyman et al. [Wyman03]. In addition to the normal shadow map, a *penumbra map* is generated by drawing shaded cones and sheets (see Section 3.2.3) for each of the occluder's edges and corners into this map. The penumbra map is then used as a projective texture and can therefore also be applied for non-planar receiver geometry.

In Part III of this thesis we will present two hardware-accelerated techniques that approximate penumbra regions using only a small number of samples on the light source.

3.2.5 Shadow Volumes

Crow's shadow volume algorithm [Crow77] is one of the most popular algorithms for shadow generation. By extending occluder polygons to form semi-infinite volumes, so called shadow volumes, shadowed pixels can be determined by simply testing if the pixel lies in at least one shadow volume. A hardware-accelerated implementation of Crow's shadow algorithm was later proposed by Heidmann [Heidmann91]. Especially for real-time applications it is the de-facto standard way for precise, high quality shadows. This is due to the fact that shadow information is generated in object space, meaning that shadow information is available for every window-space pixel.

The shadow volumes algorithm starts with the detection of possible silhouette edges. For simplicity, we assume that all shadow casting objects are closed triangular meshes (2-manifold) for which connectivity information is available.

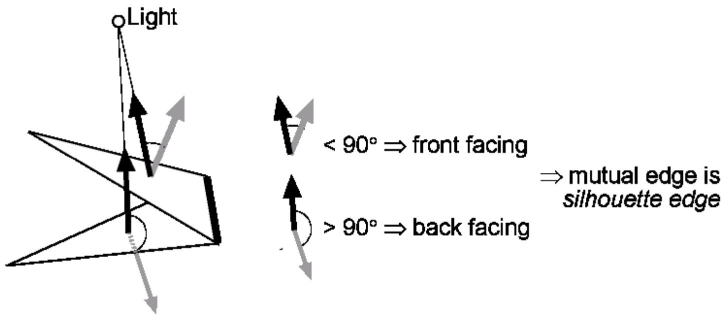


Figure 3.10: Silhouette edge detection.

To test whether a given edge is a silhouette edge we check if the edge connects a front- and a back-facing triangle, with respect to the light source. This is illustrated in Figure 3.10. Triangle orientation can easily be checked by taking the dot product of the face normal and the vector to the light source. If this dot product is negative, a triangle is back-facing with respect to the light, otherwise it is front-facing. Repeating this for all edges, we obtain a set of silhouette edges that form closed loops.

Next we extrude these silhouette loops to form semi-infinite volumes. For each silhouette edge a quadrilateral is constructed by taking the two original vertices of the edge and two vertices which are computed by moving the original vertices far away to infinity along the ray originating from the light source through the vertex.

Together with the object's front facing triangles, these quadrilaterals bound all regions in space which are in shadow. In order to check if a given point is in shadow all we have to do is to determine if the point lies outside of all shadow volumes.

This information can be easily obtained by following a ray from the viewer to the surface point and counting how many times we enter or leave a shadow volume boundary polygon. This counting scheme is illustrated in Figure 3.11.

Here shadow volumes have been generated for a sphere and a box illuminated by a point light source. While following the ray from the viewer to surface point *A*, we count how many times we enter (increment) and leave (decrement) a shadow boundary. The final counter value of 0 indicates that the surface point is lit by the

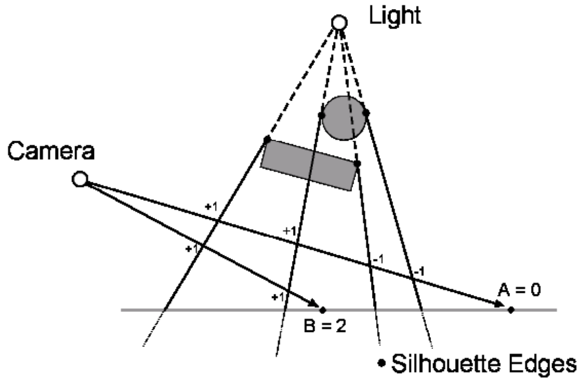


Figure 3.11: Inside-outside test.

light source, since we have left the shadow regions as many times as we entered them. Counting shadow boundaries for surface point *B* yields a value of 2, since the point is inside two shadow volumes (sphere and box).

Implementing this test using ray tracing would be a very time consuming task. Heidmann showed that this simple in-out counting can be performed on graphics hardware using the stencil buffer. First, the stencil buffer is initialized to zero (all pixels lit). Next, the whole scene is drawn as seen by the camera. In this step only depth information is relevant so color channels and all lighting and shading computations can be disabled.

The actual counting can then be achieved by disabling depth buffer writes and rendering all shadow volume quadrilaterals. In this step the stencil operation is setup in such a way that front-facing quadrilaterals (with respect to the viewer) increment the stencil value at the window space position for all pixels that pass the depth test. Similarly, all pixels that pass the depth test and belong to a back-facing quadrilateral will decrement the stencil value. On modern graphics hardware this can be implemented in a single rendering pass (two-sided stencil testing) whereas on older hardware separate passes for front- and back-facing quadrilaterals are needed (single stencil operation). Changing the counter value based on the front- and back-facing information requires a consistent winding order when constructing the quadrilaterals. This can for instance be achieved by sticking to the vertex order of the front-facing triangle (with respect to the light source) adjacent to the silhouette edge.

In a final step, the scene is rendered once again, this time with lighting and shading turned on. During rendering we set up the stencil test such that only those pixels whose corresponding stencil value is zero will pass through.

Figure 3.12 shows an example scene with shadows computed using stencil shadow volumes. As can be seen in the left image, shadow volumes generate very precise shadows for arbitrary (polygonal) receiver and occluder geometry. Figure 3.12(b) shows the scene with the silhouette edges emphasized. From these edges, shadow volumes are generated as depicted in Figure 3.12(c).

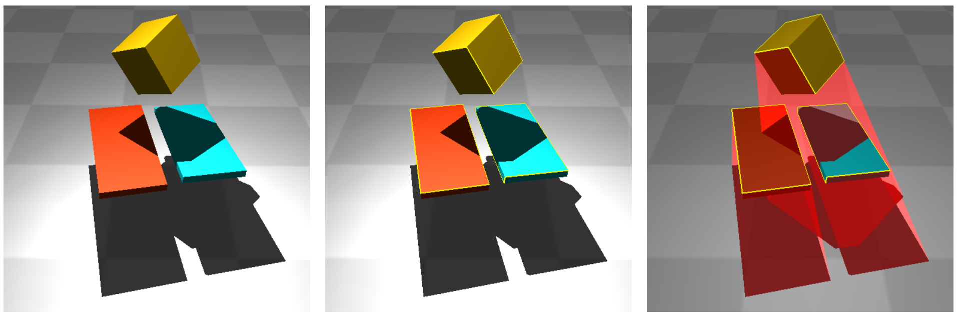

(a) final image (b) silhouette edges (c) shadow volumes

Figure 3.12: Shadow volumes.

Optimizing Silhouette Detection

With complex geometric scenes, the computation of object silhouettes can be a quite expensive task. Especially in fully dynamic environments, the silhouette edges may change from frame to frame. There a number of methods that deal with this problem.

A spatial data structure for shadow volumes was introduced by Chin and Feiner [Chin89]. They modified the well-known *binary space partitioning* (BSP) scheme, so that for every light source a BSP tree is generated that represents the shadow volume caused by the polygons facing towards the light. The algorithm was further improved by Chrysanthou and Slater [Chrysanthou95] to handle dynamic scenes as well.

McCool [McCool00] presented an algorithm that reduces the often problematic geometry complexity of Crow's method by reconstructing shadow volumes from a sampled depth map. He uses an edge detection algorithm to obtain the discontinuities in the depth map, which then act as the boundary polygons of a shadow volume. With this method an optimal shadow volume (non intersecting volumes) can be generated. However, the quality of the generated shadows may suffer from sampling artifacts due to the limited depth map resolution.

Pop et al. [Pop01] explained how the *dual space* of a primal space can be used to detect silhouette edges. In dual space, the light position becomes a plane, whereas edges stay edges. Those edges that intersect the plane are detected silhouette edges. Although this method sounds complicated at a first sight, it can greatly take advantage of temporal coherence, e.g. for moving light sources.

As graphics hardware becomes faster and faster, especially in terms of fill rate and vertex processing, a simple brute force approach is becoming popular again. Instead of detecting silhouette edges, one can also consider individual triangles as shadow casting objects and generate a shadow volume for each. This method is simple to implement but is only useful for very few, coarsely tessellated objects.

In Chapter 9 of this thesis we will present a fully hardware-accelerated implementation for detecting silhouette edges and extruding shadow volumes.

Generating and Rendering Shadow Volumes

There are a number of situations where the generation and rendering of shadow volumes is problematic and may lead to artifacts in the generated shadows.

Up to now we assumed that all shadow casting objects are well-modeled (2-manifold). For non-closed objects, the stencil counting scheme may fail due to open edges in the occluder for which no corresponding in/out side plane exists. Bergeron [Bergeron86] presented a version of Crow's algorithm which also is capable of handling non-closed objects as well as non-planar polygons. Here different increment/decrement values are used if the shadow quadrilateral is caused by an open or closed edge.

Recently, an alternative to stencil-based counting using alpha blending was proposed by Roettger et al. [Roettger02]. They replace increment and decrement operations by multiplications, which under certain conditions is even faster than using the stencil buffer. The main benefit of their method is that it works for graphics cards that do not support hardware-accelerated stencil testing.

Stencil counting may also produce wrong results due to view frustum clipping. Consider the case depicted in Figure 3.13(a). Here the shadow volume intersects the near clipping plane, meaning that some parts of the side planes, and therefore some stencil events, are missing. This can be fixed by additionally drawing the

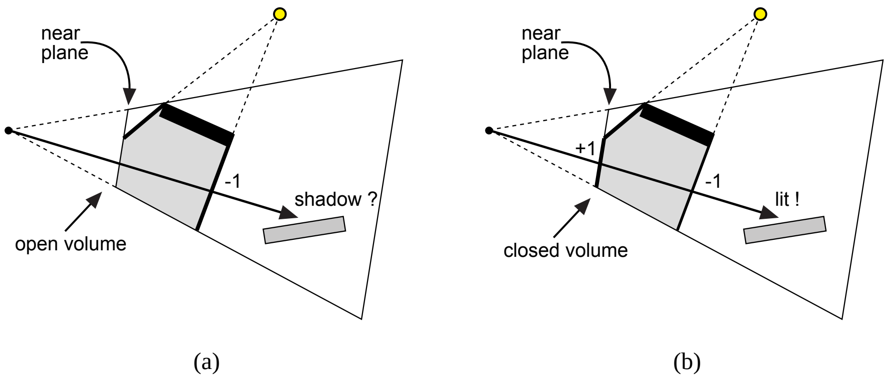


Figure 3.13: Problems due to view frustum clipping.

missing geometry at the near plane, as done by [Diefenbach94]. This preserves a closed shadow volume (Figure 3.13(b)) for which all stencil events are computed. This analytic computation can be very complicated and time consuming.

A different solution to the near plane problem has been proposed by Udeshi and Hansen [Udeshi99], where they used separate in and out counters.

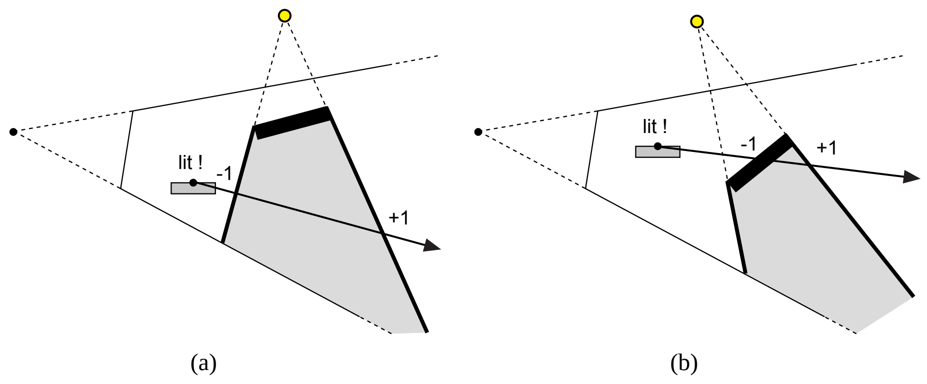


Figure 3.14: Depth-fail stencil counting.

Everitt and Kilgard [Everitt02] came up with a bullet-proof implementation of stencil based shadow volumes for hardware-accelerated rendering. They solve the problem of non-closed shadow volumes due to near/far plane clipping by inverting the stencil count scheme, an unpublished idea by Carmack [Carmack00], and moving the far plane to infinity. This way, shadow volumes are always closed, and artifact-free shadows can be generated. Figure 3.14 illustrates this method: During rendering of shadow volumes, the stencil test is setup such that it counts the number of fragments for which the depth test *fails* (in contrast to the standard depth-pass counting). Since there is no far clipping plane, it is guaranteed that all sides of the shadow volume will be counted. With depth-fail stencil testing it is necessary that all shadow volumes are completely closed, including the top and bottom. Figure 3.14(b) depicts a case in which one stencil event is due to the top of the shadow volume. Everitt showed that depth precision using a far plane at infinity is only slightly worse than a finite far plane, due to the $1/z$ quantization. The tutorial by Lengyel [Lengyel02] provides some more implementation details and shows how fill rate problems can be reduced by using attenuated light sources.

Depth-fail stencil counting also solves the problem when the viewer is inside one or more shadow volumes. With depth-pass counting, the stencil buffer needs to be initialized with the correct number, while depth-fail only counts events behind the actual pixel and therefore produces the correct result.

Soft Shadows using Shadow Volumes

Several researches adopted the shadow volume method to produce soft shadows for extended light sources.

Brotman and Badler [Brotman84] came up with a soft shadow version of Crow's algorithm where they generated shadow volumes for a number of light source samples and computed the overlap using a depth buffer algorithm.

Nishita et al. [Nishita85] constructed penumbra and umbra volumes for linear light sources. Since the analytic computation of these volumes is very expensive, the method is not suitable for a real-time implementation.

Recently, Akenine-Möller and Assarsson [Akenine-Möller02a, Assarsson03] presented a soft shadow volume method based on *penumbra wedges*. Instead of extruding a single quad for each of the occluder's silhouette edge, a wedge is constructed which represents the penumbra volume. Although the construction of penumbra wedges can be quite complicated, the rendering of wedges can be implemented using graphics hardware, which results in real-time frame rates.

Part I

Advanced Shadow Mapping Techniques for Point Light Sources

Chapter 4

Practical Shadow Mapping

In this part of the thesis we will propose a number of variants and extensions of the classical shadow mapping algorithm. We will show methods to reduce artifacts caused by the image-based nature of this approach, extensions to the type of light sources that can be used, and also methods that increase the efficiency of the shadow mapping algorithm. Since we want to use our techniques in a real-time rendering framework, all of them have an efficient hardware-accelerated implementation, which we will also discuss in detail.

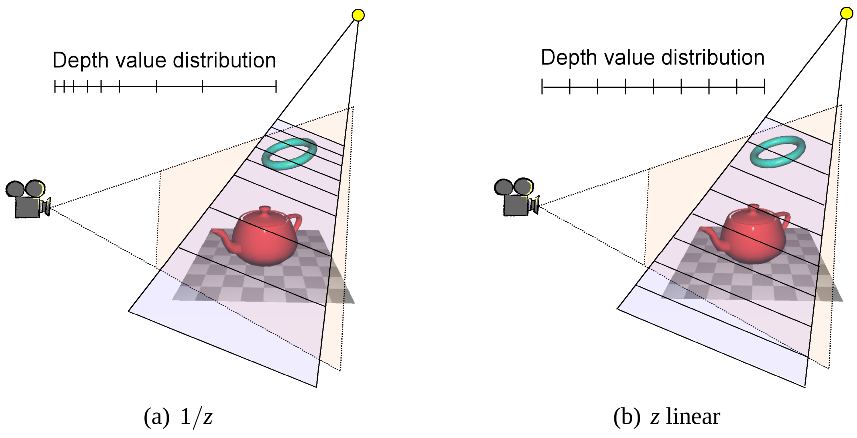
As we have seen in the previous chapter, shadow mapping has a number of drawbacks due to numerical precision and sampling. Many of these problems can be reduced if the shadow map is optimized for the current camera view and for the scene itself.

In this chapter we focus on the traditional shadow map algorithm and show how the light source's viewing frustum can be adjusted to use most of the available precision, in terms of shadow map resolution.

Since it is also important that the available depth precision is used equally for all regions inside this frustum, we first show how uniformly spaced depth values can be used when generating the shadow map. Here we present a general method that works in conjunction with dedicated shadow mapping hardware. Using color-encoded depth maps, as introduced in the previous chapter, we further improve depth precision by using an adaptive, histogram-based optimization step and Woo's method [Woo92] of using the second depth value.

4.1 Distribution of depth values

When rendering the scene from a given viewpoint, depth values are sampled non-uniformly ($1/z$) due to the perspective projection. This makes sense for the camera position, since objects near to the viewer are more important than those far away, and therefore sampled at a higher precision. For the light source position this assumption is no longer true. It could be the case that objects very far from the light source are the main focus of the actual camera, so sampling those at lower z precision may introduce artifacts, e.g. missing shadow detail. A solution to this was e.g. presented by Heidrich [Heidrich99] (Section 3.2.4). Here depth values are sampled uniformly using a 1D ramp texture that maps eye space depth values to color values, which are later used as the corresponding shadow map.



(a) $1/z$ (b) z linear

Figure 4.1: Distribution of depth values.

Figure 4.1 illustrates the difference between linear and $1/z$ mapping. On the left side, depth values are sampled using the traditional perspective projection. Objects near to the light source obtain most of the available depth values, whereas objects far away (e.g. the ground plane) have less precision. Shadow details for the teapot may be missing while the torus may be oversampled. The right side of Figure 4.1 shows the same setup using a linear distribution of depth values. Here all objects are sampled equally. We can achieve this linear distribution of depth values using a customized vertex transformation, which can be implemented using the so called *vertex shader* or *vertex program* functionality introduced in Section 2.3.2.

Instead of transforming all components of a homogeneous point $P = (x_e, y_e, z_e, w_e)$ by the perspective transformation matrix, e.g. $(x, y, z, w) = Light_{proj} \cdot P$, we replace the z component by a new value $z' = z_l * w$. The linear depth value $z_l \in [0; 1]$ corresponds to the eye space value z_e mapped according to the light source near and far plane:

$$z_l = -\frac{z_e + near}{far - near}$$

To account for normalization $(x/w, y/w, z/w, 1)$ which takes place afterwards (normalized device coordinates), we also pre-multiply z_l by w. This way, the z component is not affected by the perspective division, and depth values are uniformly distributed between the near and far plane.

4.1.1 Second Depth Shadow Maps

As already mentioned, Heidrich [Heidrich99] proposed a very clever method for shadow mapping that works with the standard OpenGL feature set. Despite the linear quantization of the depth values, numerical precision and quantization artifacts remain a critical issue. Deeper framebuffers and textures, as well as higher precision operations in the texturing units of graphics hardware will certainly help to reduce these problems. Nonetheless, some algorithmic advances to improve numerical stability are highly desirable.

Woo [Woo92] modified the shadow mapping algorithm to use a depth map where the reference value is actually a weighted sum of the visible surface and the first surface point behind it. This improved the numerical stability of the shadow map algorithm, because it mostly avoids depth comparisons of very similar values, resulting in an artifact known as *surface acne* (as shown in Section 3.2.4).

In order to implement this algorithm with OpenGL, we have to alter the method for generating the shadow map. In addition to generating a depth image of the scene that represents the visible surfaces, we now also have to generate a second depth image containing all the second surfaces. This can be achieved with an additional rendering pass using alpha blending: In a first pass, the shadow map is generated in the alpha channel as before. In a second pass, the scene is rendered again with the same texture coordinates, but this time the 1D texture also contains a linear ramp in the color channels in addition to the alpha channel. Furthermore, we disable writing to the alpha channel and the depth buffer, and set up the depth test such that only those pixels get rendered where the new depth value is larger than the value already contained in the framebuffer. This ensures, that the frontmost (i.e. visible) surfaces do not get rendered a second time. At the same time, framebuffer blending is set up to select the minimum of all color values for each pixel. This will select the frontmost surface of all those that survived the depth test, which is the second surface in total. Before these rendering passes, the framebuffer should be cleared to a value of 1, corresponding to objects further away than the far plane.

The result of this method is a framebuffer containing the depths of the visible surfaces in the alpha channel, and the depths of the second surfaces as a luminance value. We can employ a color matrix, which is part of the OpenGL imaging subset to average the two and store the result in an alpha texture. On multitexturing systems, we can also simply load the two intersections as two separate textures, and apply them independently of each other. This avoids the color matrix, which may not be available on some systems. An example for a second depth shadow map is depicted in Figure 4.2.

Contrast Improvements

Another way to make Heidrich's algorithm numerically more stable is to further improve the distribution of numerical precision across the whole depth region between near and far plane in the shadow map. For example, if the scene consists of

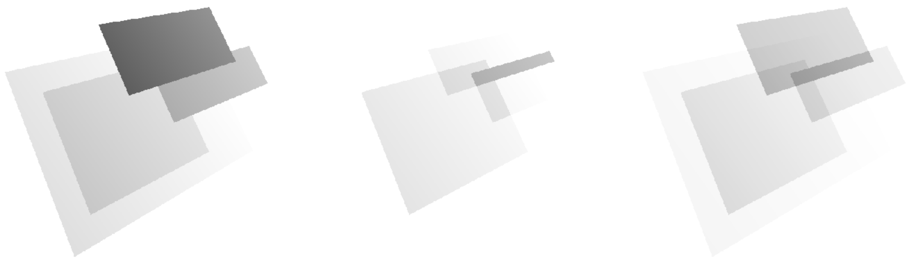

(a) first (b) second (c) average

Figure 4.2: Second depth shadow maps.

several objects that are close to the light source and several that are away from it, but only few objects in between, then a uniform quantization of the depth values as described above wastes numerical precision in this center part.

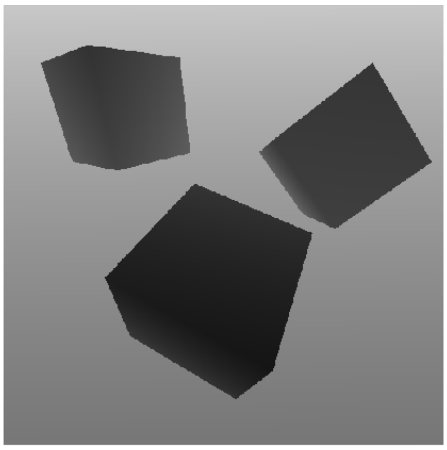
In this case it would be better to replace the linear ramp texture that acts as a table mapping from z coordinates into the interval $0 \dots 1$ by some nonlinear but monotonic function. This function should be designed to maximize the contrast in the depth map in the sense that all quantization levels should appear roughly equally often in the resulting depth map. In image processing, this task is known as histogram equalization, and algorithms for determining an adequate table given a histogram of the original, uncorrected image are well known [Gonzalez92].

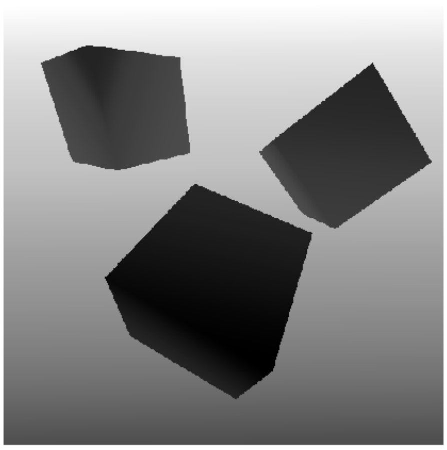
Thus, in order to perform a histogram equalization on the shadow map, we need to acquire the histogram of the original depth map. We can do this either using software, or the histogram feature provided by the OpenGL imaging subset. Ideally, we would then for every frame have to generate the original shadow map, compute the histogram, and from it the table that equalizes the histogram, and finally re-render the shadow map using the computed table. In an animation, where light sources and objects move gradually, we can, however, get rid of one of these passes if we are willing to work with the table of the preceding frame.

We then simply generate the shadow map using the old table from the preceding frame, and, while this map is transferred to texture memory, we let OpenGL compute its histogram. That histogram is then used to generate a new table which we then use in the next frame. Figure 4.3 shows the effect of histogram equalization on a depth map.

4.1.2 How near, how far ?

A very important property that affects the depth precision of the shadow map is the setting of the near and far plane when rendering from the light source position. A common approach is to set those to nearly arbitrary values like 0.01 and 1000.0 and hope that a shadow map with 24 bit precision will still cover enough of the


(a) original map


(b) normalized map

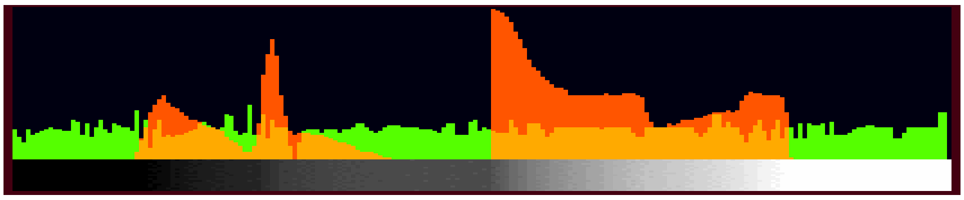

(c) histogram (red: original *z* distribution, green: optimized *z* distribution)

Figure 4.3: Histogram equalization.

relevant shadow information.

By analyzing the scene we can improve depth precision by setting the near and far plane such that all relevant objects are inside the light's viewing frustum, as depicted on the left side of Figure 4.4.

In terms of depth precision, this setting is still far from being optimal. It is clear that the torus needs to be included in the shadow map, since it will cast a large shadow onto the ground plane and teapot, but for this shadow information only one bit would be sufficient since the shadow caster itself is not seen by the camera. So what we really would like to have is some kind of tight fitting near and far plane setup that concentrates on those objects that are visible in the final scene (seen from camera position). This optimal setting is depicted on the right side of Figure 4.4. If we would render this scene with the traditional approach, shadows cast by the torus would be missing since the whole object lies outside the light's viewing frustum and would be clipped away.

We can easily include such objects by having depth values of objects in front of the near or beyond the far plane clamped to zero or one, respectively. This clamping can be achieved with a special *depth replace* texture mode[1], available

[1]As the name implies, this texture mode replaces a fragment's window space depth value.

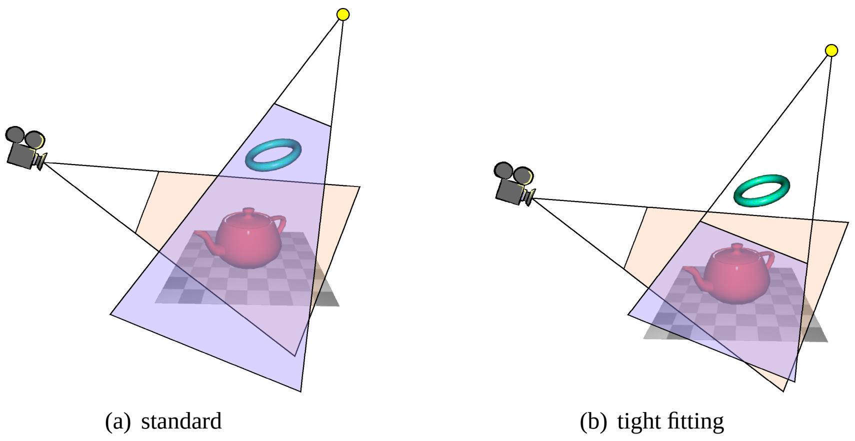

(a) standard (b) tight fitting

Figure 4.4: Setting of near and far plane.

as part of the *texture shader* extension [NVIDIA02] provided by recent NVIDIA graphics cards.

Assume we want to render a shadow map with 16 bit precision where depth values outside the valid range are clamped rather than clipped away. These depth values can be encoded using two bytes, where one contains the least significant bits (LSB) while the other stores the most significant bits (MSB). If we setup a two dimensional ramp texture in which we encode the LSBs in the red channel (0 to 255, column) and in the green channel we store the MSBs (0 to 255, row position), we can map the lower 8 bit of a given z value by setting the s coordinate to $256.0 * z$ and using the s coordinate repeat mode. This way s maps the fractional part of $256.0 * z$ to the LSB entry in the ramp texture. To code the MSB we can directly map z to t and use a *clamp-to-edge* mode such that values $z < 0$ are clamped to 0 and values $z > 1.0$ are clamped to 1.0.

To replace the current window space depth value with the new, clamped value we now have to setup the texture shader as depicted in Figure 4.5. Texture unit 0 is responsible for mapping the higher and lower bits of the depth value to a color encoded RGB value (blue component set to zero). Texture unit 1 is configured to perform a *dot product depth replace* operation, which takes texture coordinates from unit 1 and computes a dot product with the result of the previous texture unit (color encoded depth). The result is a new depth value that is just a clamped version of the original depth value.

One problem with this texture shader setup is that the LSB is repeated even for objects in front or beyond the near/far planes, due to the s coordinate texture repeat. If we set the planes such that all pixels in front of the near clipping plane are mapped to a MSB of 0 and pixels beyond the far clipping plane to a MSB of 255 we do not have to worry about the LSB part of the depth value. So the effective range of depth values is between 0x0100 and 0xfeff. This method can be extended

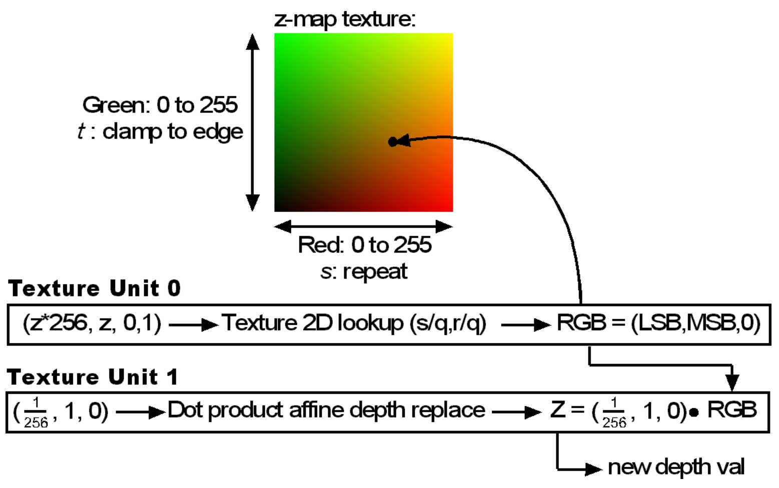

z-map texture:
Green: 0 to 255
t : clamp to edge
Red: 0 to 255
s: repeat
Texture Unit 0
(z*256, z, 0,1) → Texture 2D lookup (s/q,r/q) → RGB = (LSB,MSB,0)
Texture Unit 1
(1/256, 1, 0) → Dot product affine depth replace → Z = (1/256, 1, 0)• RGB
new depth val

Figure 4.5: Texture shader setup.

to 24 bit depths by using a special HILO texture format [NVIDIA02], for which filtering takes place at 16 bits per component.

Up to now we did not take care about the view frustum culling that takes place before the rasterization. If for example an object lies completely in front of the near clipping plane all triangles would be culled away after the transformation step (clip coordinates). To avoid this we simply modify the vertex shader described in Section 4.1 such that the z component of the output position is set to a value of $0.5 * w$. This way all vertices are forced to lie between the valid $[0;1]$ z range. The z values passed as texture coordinates for texture unit 0 are still the linear z_l's. After the depth replace step we then restore valid z coordinates used for depth testing.

4.2 Concentrating on the visible part

In the previous section we discussed how important the setting of near and far clipping is with respect to the depth resolution. For the shadow map resolution (width and height) the remaining four sides of the light's viewing frustum are crucial.

Consider a very large scene and a spotlight with a large cutoff angle. If we would just render the shadow map using the cutoff angle to determine the view frustum we would receive very coarse shadow edges when the camera focuses on small portions of the scene. Hence it is important that the viewing frustum of the light is optimized for the current camera view. This can be achieved by determining the visible pixels (as seen from the camera) and constructing a viewing frustum that includes all these relevant pixels.

In order to compute the visible pixels, we first render the scene from the camera position and use projective texturing to map a control texture onto the scene. This control texture is projected from the light source position and contains color-coded information about the row-column position, similar to the ramp texture used for the depth replace. In this step we use the maximal light frustum (cutoff angle) in order to ensure that all illuminated parts of the scene are processed. Since pixels outside the light frustum are not relevant we reserve one row-column entry, e.g. $(0,0)$, for outside regions and use this as the texture's border color. By reading back the framebuffer to host memory we can now analyze which regions in the shadow map are used. In the following subsections we will discuss methods for finding a suitable light frustum based on this information.

4.2.1 Axis aligned bounding rectangle

The easiest and fastest method is to compute the axis aligned bounding rectangle that encloses all relevant pixels. This can be implemented by searching for the maximum and minimum row and column values, while leaving out the values used for the outside part (texture border). This bounding rectangle can now be used to focus the shadow map on the visible pixels in the scene. All we have to do is to perform a scale and bias on the x and y coordinates after the light's projection matrix to bring

$$[x_{min};x_{max}] \times [y_{min};y_{max}] \rightarrow [-1;1] \times [-1;1] \qquad .$$

4.2.2 Optimal bounding rectangle

A better solution for adjusting the view of the light source is to compute the optimal bounding rectangle that encloses all visible pixels. This can be realized by using a method known as the *rotating calipers* algorithm [Toussaint83, Pirzadeh99] which is capable of computing the minimum area enclosing rectangle in linear time. We start by computing the two dimensional convex hull of all visible points using the *monotone chain* algorithm presented by [Andrew79].

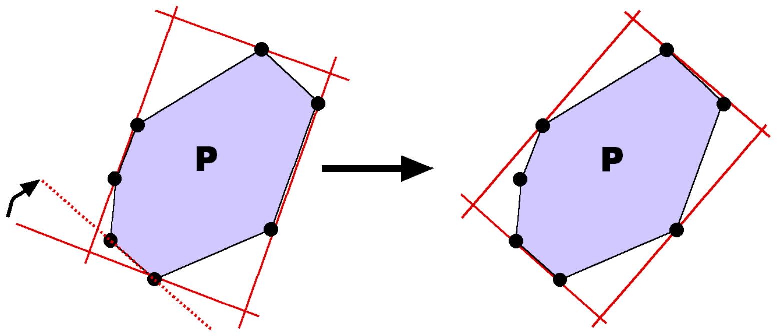


Figure 4.6: Rotating calipers.

As stated by [Pirzadeh99], the minimum area rectangle enclosing a convex polygon P has a side collinear with an edge of P. Using this property, a brute-force approach would be to construct an enclosing rectangle for each edge of P. This has a complexity of $O(n^2)$ since we have to find minima and maxima for each edge separately. The rotating calipers algorithm rotates two sets of parallel lines (calipers) *around* the polygon and incrementally updates the extreme values, thus requiring only linear time to find the optimal bounding rectangle. Figure 4.6 illustrates one step of this algorithm: The support lines are rotated (clockwise) until a line coincides with an edge of P. If the area of the new bounding rectangle is less than the stored minimum area rectangle, this bounding rectangle becomes the new minimum. This procedure is repeated until the accumulated rotation angle is greater than 90 degrees [2].

4.2.3 Examples

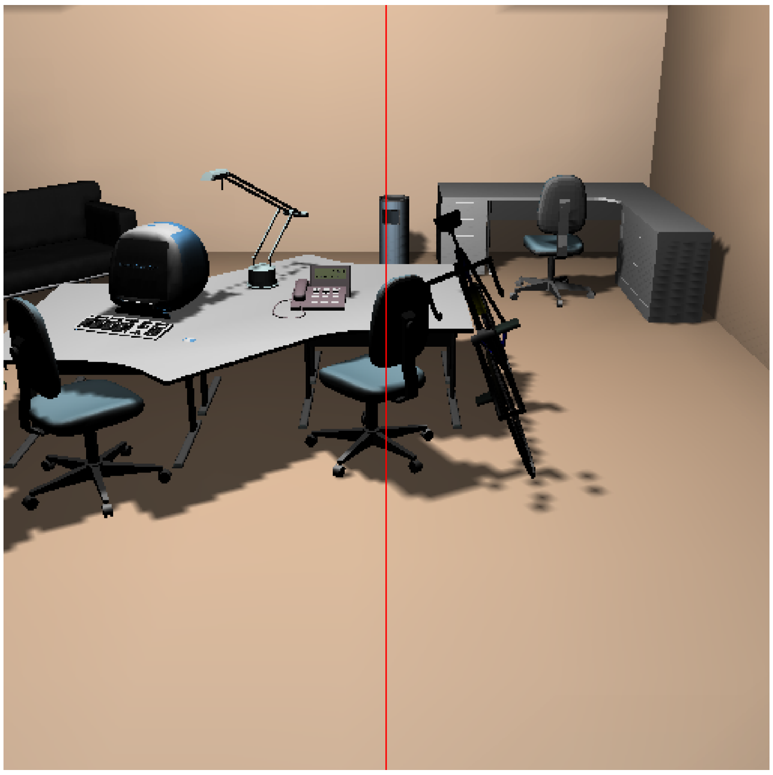
Figure 4.7 shows an example scene illuminated by one spotlight with a large cutoff angle. Here, the image resolution was set to 512×512 pixels (4-times oversampling), whereas the shadow map only has a resolution of 256×256 pixels. For the control rendering pass a 64×64 pixel region was used. The frame rates for this scene are about 20 to 25 frames per second (74000 triangles).

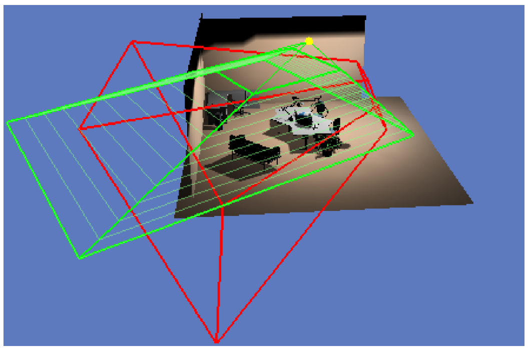
In Figure 4.7(a) we directly compared the adjusted light frustum (left half of the image) with the result obtained using some fixed setting (right half). Here the adjustment can only slightly improve the shadow quality (coarse shadow edges), but the algorithm still computes an optimal setting for the light's near and far clipping plane.

Figure 4.7(b) shows the optimized light frustum and the camera frustum, seen from a different perspective. In Figure 4.7(c) the convex hull and the resulting minimum area enclosing rectangle are drawn as they are located in the non-optimized shadow map.

In Figure 4.8(a) the camera was moved so that it focuses on a small part of the scene. Here the automatic adjustment greatly improves shadow quality since the shadow map now also focuses on the important part of the scene. Figure 4.8(b) shows the normal depth map, as it would look with an un-optimized frustum, whereas Figure 4.8(c) shows the depth map as it is generated with our method. It can be seen that the light frustum is slightly over-estimated. This is due to the resolution of the control texture. An even tighter fit can be achieved by repeating the control texture rendering several times, so that the frustum converges near the optimum.

[2] For a detailed description of the algorithm please see [Pirzadeh99].


(a) adjusted vs. normal


(b) frustum settings

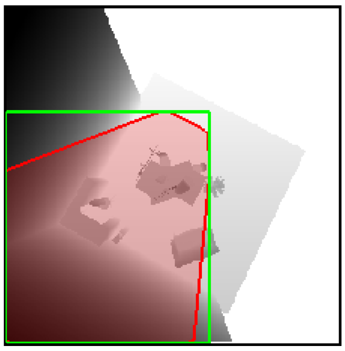
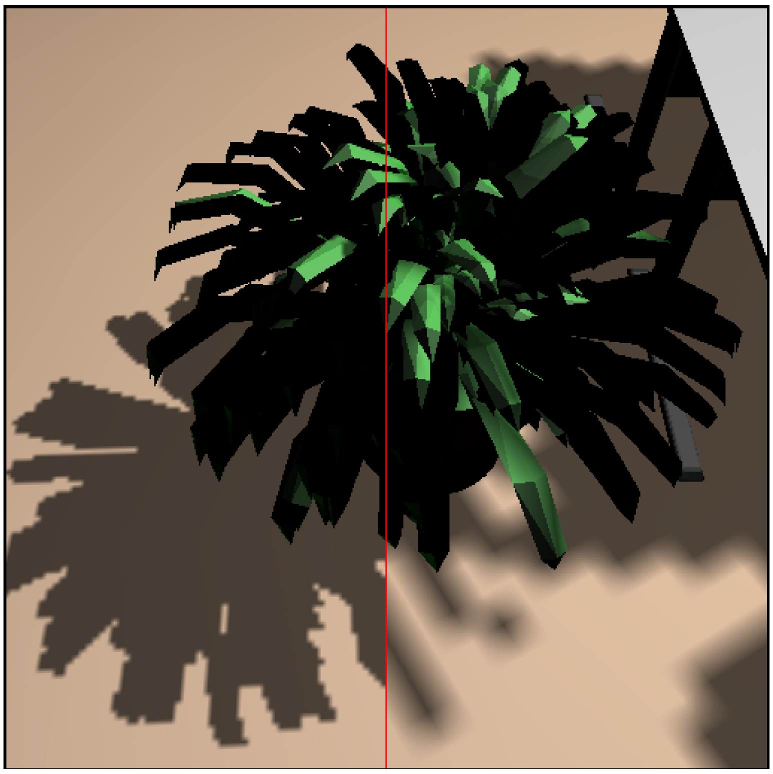
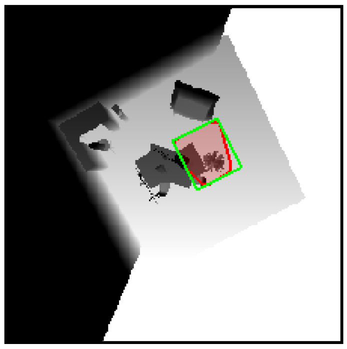

(c) depth map

Figure 4.7: Scene rendered with light frustum adjusted.


(a) adjusted vs. normal


(b) normal depth map

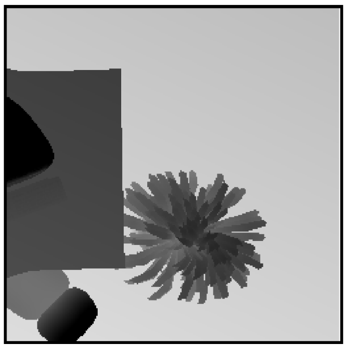

(c) focused depth map

Figure 4.8: Camera close-up.

4.3 Discussion

In this chapter we have shown how the accuracy of shadow mapping can be greatly enhanced using a light source frustum that adapts to the actual camera view and depth values that are uniformly spaced between the light source's near and far plane. Since we clamp depth values rather than clipping geometry we can include shadow casters that are not visible from the camera view without stretching the light frustum, thus restricting depth precision to the visible part. On recent graphics cards, depth clamping can also be implemented using fragment programs (Section 2.3.3), instead of the texture setup proposed here. Recently, some cards also directly support depth clamping, e.g. NVIDIA's GeForceFX.

The tight fitting light frustum is not only useful for shadow maps based on William's original work. When using e.g. *Perspective Shadow Maps*, as presented by Stamminger and Drettakis [Stamminger02], we can also concentrate this type of shadow map on the visible and illuminated parts of the scene, which should improve the sampling ratio even further.

Using color-encoded depth values, precision can be improved by using histogram equalization, which concentrates depth samples to regions where they are actually needed. Also the often problematic user-defined shadow test bias is not needed when using the average depth of the first and second depth values.

Chapter 5

Shadow Map Filtering

The previous chapter proposed several techniques that can be used to adjust the shadow map to the actual scene parameters and enhance the distribution of depth values. However, shadow mapping suffers especially from undersampling artifacts when it comes to higher frequency parts, a problem that was not addressed so far.

As explained in Section 3.2.4, Reeves et al. [Reeves87] proposed a way of filtering shadow maps using so called *percentage closer filtering* (PCF), which basically performs several shadow tests for each pixel and outputs the percentage visibility value, based on the outcome of the shadow tests. Recently, some graphics cards directly support PCF, which greatly enhances shadow quality when using shadow maps in real-time rendering.

In this chapter we will show how Reeves' approach can also be implemented in hardware supporting only the standard OpenGL feature set. Our algorithm therefore relies on Heidrich's [Heidrich99] color-encoded depth maps, introduced in Section 3.2.4.

Besides Reeves' original idea, we also present a variant called *Fast PCF*, which is especially suitable for real-time applications.

5.1 Hardware-based Percentage Closer Filtering

A special PCF implementation can be realized by extending the color-ramp shadow mapping technique. In Heidrich's approach, only one shadow test for each fragment is performed, which will result in blocky shadow edges when the shadow map resolution is too low. In order to filter the shadow map, we would need to calculate the area occupied of the projected pixel in the shadow map, and perform several shadow tests over this region, which is not possible on standard OpenGL hardware.

However, it is possible to use PCF if we assume a constant filter region, e.g. a 2×2 footprint, which is the smallest, symmetric filter size. For this we have to compare each pixel's z against four z values stored in the shadow map.

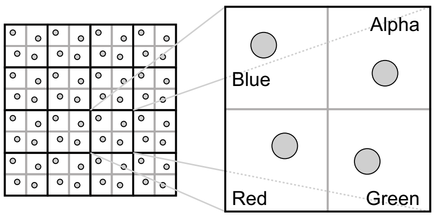


Figure 5.1: Stratified sampling and pixel packing.

For a constant footprint we are able to generate a shadow map where each entry consists of *n* components and where *n* is the number of samples per footprint. Given a 2×2 footprint the four components can simply be stored using the red, green, blue, and alpha channel of the texture image. For the generation of the shadow map this means that we have to render the scene four times where in each pass only one color channel is enabled for writing and the image plane is jittered as depicted in Figure 5.1. This stratified sampling scheme increases the effective resolution by a factor of two in each dimension, so instead of a 1024×1024 one component shadow map we have now generated a 1024×1024 shadow map with four depth values per texel.

Given such a packed shadow map it is relatively easy to adapt Heidrich's alpha-based shadow test since we only have to extend the one component scheme to four components (RGBA).

During the shadow test passes we first render the scene as seen by the camera but since we want to compare the surface point's depth against every component in the shadow map we replicate the z values over all four color channels. Next, the projection of the shadow map is done as before. This corresponds to Reeves' comparison step, resulting in a four component shadow mask.

In order to compute one scalar value per pixel which should represent the percentage shadowing, we have to count the non zero values in the shadow mask and divide this sum by the number of samples taken per pixel. All this can be performed within a single framebuffer to framebuffer copy.

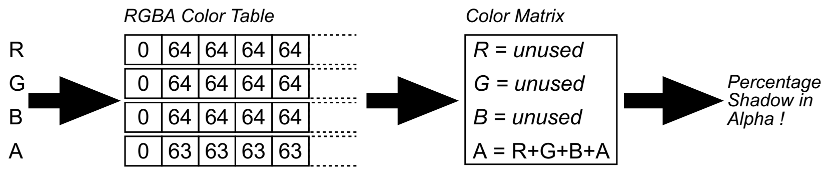


Figure 5.2: Computing the percentage of shadowing.

Assuming a color depth of 8 bits per component, we setup the OpenGL imaging pipeline as depicted in Figure 5.2. At first, incoming values are transformed using a RGBA color table. This clamps values to either 0 (completely lit) or 64, which represents 25% shadow in total[1]. Second, a simple 4×4 color matrix is used to sum up the contributions of all color channels and to pass out the result as the new alpha value. After this, five different levels of shadowing per pixel are stored in the alpha channel of the framebuffer: 0% shadowed (lit), 25%, 50%, and 75% for partially shadowed pixels and 100% for pixels that are completely shadowed.

This algorithm works very well for footprints of size 2×2 since all components can be processed simultaneously using the four color channels. If it comes to larger filter sizes, e.g. 3×3 or 4×4, the algorithm needs to be split up into parts of a maximum of four components per texel. The resulting contributions can then be summed up using the accumulation buffer [Haeberli90].

Although theoretically possible, filter sizes greater than 2×2 are no longer practical for interactive or real-time applications. Considering a filter region of 4×4, the generation of the shadow map would require 16 rendering passes and four RGBA texture maps to store the results. During the shadow test, another eight passes are necessary to perform the subtraction plus up to four framebuffer copies and accumulation buffer operations.

5.2 Fast PCF for Real-Time Applications

In Section 5.1 a hardware-based algorithm for percentage closer filtering using Reeves' original method was proposed. However, for real-time applications this method requires way too much hardware resources, especially when it comes to larger filter sizes. In this section we present a slightly modified version of Reeves' PCF that overcomes these limitations.

Considering the generation of the shadow map, a footprint of $n \times n$ reduces the effective resolution by a factor of n in each dimension. For hardware graphics this means that we either have to use a very large shadow map or need to render the scene several times to different color channels. Since the number of rendering passes and the image resolution are critical for real time frame rates, *real* PCF is not well suited for very complex and dynamic scenes or machines with limited hardware resources.

A faster way of performing percentage closer filtering can be achieved if we try to retain the effective resolution of the shadow map and use a filtering scheme that softens shadow boundaries by just looking at adjacent texels to compute the shadow mask.

To do this, we render the shadow map using a 1D color ramp texture (as explained in Section 3.2.4), but instead of using only one color channel we store the encoded depth values in all four channels. Next, we want to generate a packed shadow map where each texel consists of four adjacent pixels as shown in Figure 5.3. Collecting neighboring pixels in that manner is not a trivial task since most

[1]Using a value of 63 in one of the color channels ensures that the value for completely shadowed pixels sums up to 255.

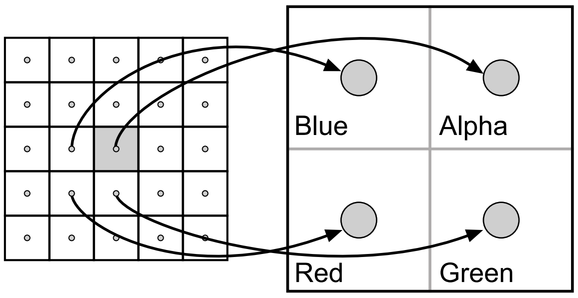


Figure 5.3: Fast PCF filtering and pixel packing.

rasterization hardware does not provide efficient methods for changing the position of pixels[2]. One exception to this is the OpenGL Imaging Subset, which does provide a method for performing convolutions on image data in either one or two dimensions. In the case of a 3×3 RGBA convolution the new color of a pixel P' is computed as

$$P'_{ij} = \sum_{m=0}^{2}\sum_{n=0}^{2} C_{mn}P_{m+1,n+j} \quad , \tag{5.1}$$

where C is the 3×3 RGBA convolution filter and P a specific pixel in the input image. This weighted sum can be adapted to perform the pixel packing as depicted in Figure 5.3 if we setup the filter kernel to collect only one color component for each color channel. Using

$$C = \begin{pmatrix} [0,0,0,0] & [0,0,0,0] & [0,0,0,0] \\ [0,0,\mathbf{1},0] & [0,0,0,\mathbf{1}] & [0,0,0,0] \\ [\mathbf{1},0,0,0] & [0,\mathbf{1},0,0] & [0,0,0,0] \end{pmatrix}$$

the resulting pixel value P' consists of the red channel taken from the lower left, the green channel from the lower mid pixel and so on. Although one column and one row of the filter kernel is not used at all, we prefer filter sizes with odd widths and heights, e.g. 3×3 or 5×5, since graphics hardware is normally optimized for this kind of filter sizes[3].

After this convolution, which is applied when we copy the framebuffer contents to the RGBA shadow map, we have packed four depth samples into a single texel. This differs from the pixel packing scheme presented in Section 5.1 since the resolution remains constant (using only one rendering pass for the shadow map).

Performing the shadow test and computing the percentage shadowing term is very similar to the hardware-based PCF algorithm in Section 5.1, except that the texture coordinates need to be slightly offset by $(\frac{ds}{2}, \frac{dt}{2})$ (see Figure 5.4) to account

[2]Apart from some global methods, e.g. for scaling image data.

[3]This is due to the fact that most image processing convolutions, e.g. Gaussian blur, are symmetric and pixel centered.

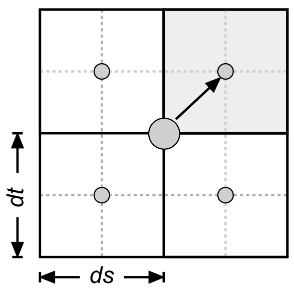


Figure 5.4: Texture coordinate offset.

for the new center pixel (since the convolution filter packs the lower left part as the new center pixel, as illustrated in Figure 5.3).

Having only one rendering pass for the shadow map generation, it becomes affordable to use larger filter sizes. In the case of a 4$\times$4 footprint we can split up the computation into four shadow mapping phases and use the accumulation buffer to sum up the results. For each pass we use a 3$\times$3 convolution that samples either the upper left, upper right, lower left, or lower right 2$\times$2 region as explained before. With this multipass method, $4*5 = 20$ shadowing levels can be generated.

5.3 Results

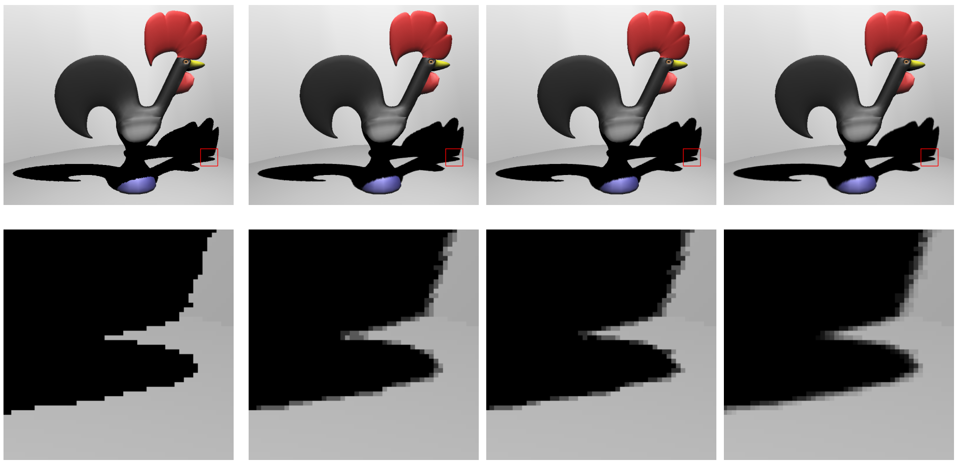
We have implemented the described methods on Silicon Graphics Octane VPro/8 and O2 workstations using OpenGL as an underlying graphics API. Since the execution time of the algorithms depends on high polygon throughput (rendering the scene several times from different points of view) and high fill rates (framebuffer and texture map copies) this kind of machines, which are optimized for both classes of applications, are ideal platforms. Furthermore, a hardware-accelerated OpenGL imaging pipeline, needed for the fast PCF algorithm presented in Section 5.2 is only supported on mid- and high-level graphics workstations.

A comparison of the different variants of shadow filtering techniques is depicted in Figure 5.5. In order to make differences more noticeable, a small part (red rectangle) of each image is magnified. The scene consists of about 7000 polygons and was rendered using an image resolution of 800$\times$600 pixels with normal OpenGL lighting and one light source enabled.

Starting from left to right, the first image shows the result without shadow map filtering. Using a shadow map resolution of 512$\times$512 pixels, undersampling artifacts at shadow boundaries are quite noticeable (blocky edges). On an SGI Octane VPro/8, this scene can be rendered at about $25-30$ frames per second[4].

The second column shows the same scene but this time with percentage closer filtering applied as described in Section 5.1. With only three more grey levels,

[4]All times presented here include the generation of the shadow map (as in fully dynamic scenes).



(a) without filtering (b) normal PCF, 2×2 (c) fast PCF, 2×2 (d) fast PCF, 4×4, multipass

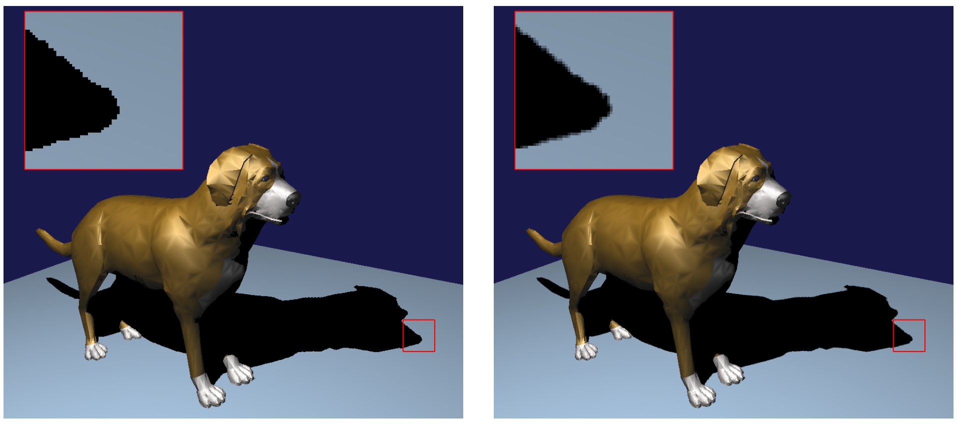
Figure 5.5: Comparison of shadow filtering techniques.

the shadows look much more realistic. The shadow map still has a resolution of 512×512, but since we used a 2×2 filter, which requires four rendering passes during the shadow map generation phase, we virtually increased the resolution by packing four depth values into a single texel. Frame rates using this method drop down to about 10 fps, which is due to the three additional rendering passes needed for shadow map generation.

With fast percentage closer filtering enabled (Section 5.2), we can achieve interactive frame rates of about 15 – 20 fps. Using a 2×2 footprint combined with the modified pixel packing method, shadow boundaries are well smoothed (Figure 5.5(c)) resulting in an image quality comparable to the normal PCF method.

The last column shows the result of fast percentage closer filtering using a 4×4 footprint. Having about 20 different levels of shadowing, blockiness is very much reduced. As described in Section 5.2, filters of sizes larger than 2×2 need to be implemented using multipass rendering and an accumulation buffer to sum up the results. Due to this, a frame rate of only 5 fps can be achieved. If we restrict ourselves to stationary lighting, the shadow map generation becomes a precomputing step which makes PCF with filter sizes larger than 2×2 affordable.

Figure 5.6 shows another example scene. On the left side the scene was rendered without filtering. Although the shadow map resolution was increased to 1024×1024 pixels, the shadow boundaries are still very blocky. Using fast PCF with a filter size of 2×2, shadow boundaries appear well smoothed (right). The rendering times are about 15 versus 10 frames per second.



(a) without filtering (b) fast PCF, 2×2 filter

Figure 5.6: Test scene.

5.4 Discussion

In this chapter we showed how Reeves' percentage closer filtering can be applied for hardware-based shadow map rendering. With this approach, shadows of high quality can be rendered at interactive or real-time frame rates. As the algorithm makes intensive use of the OpenGL imaging extensions, a hardware-only implementation is currently only possible for certain graphics workstations. For consumer-class PC graphic cards additional memory transfers from frame buffer to host (and back) are necessary to emulate imaging operations in software.

When discussing the algorithm we did not address sampling artifacts due to the limited depth resolution. When encoding depth values as color values we loose a lot of precision because of the (normal) 8 bits per color channel. A solution to this is already possible since some architectures, e.g. SGI's Octane VPro or InfiniteReality, support color depths of 12 bits per channel. Recent PC-class graphics hardware supports offscreen buffers with even floating point precision, but do normally not support a fully hardware-accelerated implementation of the OpenGL imaging subset. As shown in the previous chapter, we can enhance depth precision by using the histogram refinement or Woo's second depth idea. Combining these method with the PCF algorithm proposed here allows us to use high-quality shadow mapping on graphics hardware that does not directly support it.

Chapter 6

Dual-Paraboloid Shadow Mapping

One main argument that often leads people to prefer shadow volumes over shadow mapping is the limited *field-of-view* when generating a suitable shadow map for a given point light source. In cases where a point light source radiates over the complete hemisphere or even omnidirectional, traditional shadow mapping (using only a single depth map) fails.

The only solution to this problem so far is to use more than one shadow map. In the worst case, up to six shadow maps must be used to cover the complete environment [Dietrich01]. Although these maps can be represented as a cube map (resulting in a single texture lookup when performing the shadow test), this approach still requires up to six rendering passes when generating the shadow maps, making it unsuitable for interactive or real time applications.

In this chapter we present a method to perform shadow mapping for hemispherical and omnidirectional point light sources using only one (hemispherical) or two (omnidirectional) rendering passes for the generation of the shadow map and one final rendering pass to perform the shadow test. Similar to the traditional shadow mapping technique, all steps of the algorithm can be implemented using graphics hardware.

6.1 Shadow Mapping for Hemispherical and Omnidirectional Light Sources

Shadows generated with the traditional shadow mapping algorithm are limited to the view frustum used when generating the shadow map, as depicted in Figure 6.1(a).

Since all relevant occluder geometry is inside the frustum shadows are generated as expected. This setup is useful if the light source has a spotlight characteristic (only objects inside the spotlight cone are illuminated) but fails if the light

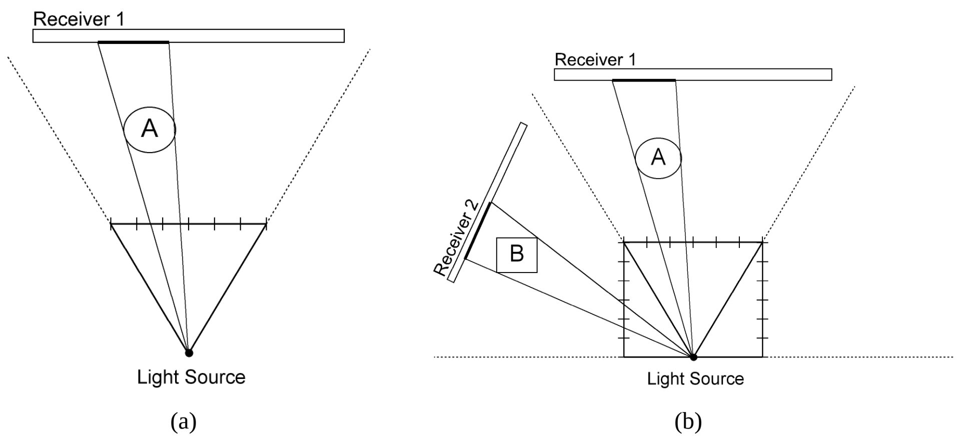


Figure 6.1: (a): Traditional shadow mapping frustum. (b): Multiple shadow maps for field-of-view of 180° (hemispherical).

source is hemispherical or omnidirectional (*field-of-view* $>=$ 180°). The trivial solution to handle such cases would be to use a number of shadow maps that together cover the whole environment seen by the light source.

Figure 6.1(b) shows such a setup for a light source with a viewing angle of 180°(hemispherical). In order to cover Object A as well as Object B the environment needs to be subdivided in a cube like manner where a separate shadow map is used for each side of the cube[1]. While this setup can be efficiently rendered if the graphics hardware supports cube map textures (as explained in [Dietrich01]) the generation of the (up to six) shadow maps is still too expensive.

The solution to this problem is quite simple: In order to minimize the cost of generating shadow maps we have to find a way of computing a shadow map that covers the whole field-of-view of the given light source. Since the sampling rate should be somehow constant to avoid changes in shadow quality we have to choose a parameterization that fulfills this criterion and that is also easy to compute (hardware-accelerated rendering).

We can find such parameterizations in the field of environment mapping, which approximates global illumination by precomputing a so called *environment map* [Blinn76] which is later used to determine the incoming light for a given direction (reflection vector). Although there exist a vast number of 3D-to-2D mappings (used to *flatten* a panoramic environment into a 2D texture map), only a small subset is really appropriate for hardware accelerated rendering:

Spherical Mapping. For a long time this was the *standard* parameterization used to represent environment maps [Haeberli93]. This parameterization can be simple explained by imagining a perfectly reflecting mirror ball centered around the object of interest. Although this parameterization only has one

[1]Since only one hemisphere will be lit, only five sides of the cube are needed.

point of singularity, the sampling rate varies significantly since pixels get extremely distorted towards the perimeter of the flattened sphere.

Blinn/Newell Mapping. A different parameterization of the sphere was proposed by [Blinn76]. Here 2D coordinates (u,v) are computed using a longitude-latitude mapping of the direction vector. Although this approach does not introduce as much distortion as the previous sphere mapping technique, it is not commonly used due to the expensive longitude-latitude mapping (which involves computing *arctan* and *arcsin*).

Cube Mapping. As already mentioned in previous section, the cube map parameterization [Voorhies94] is very popular since it does not require any re-warping to obtain images for the cube faces. Many graphics cards (e.g. NVIDIA, ATI) directly support texture fetching using cube maps. Again the main disadvantage are the number of rendering passes during the generation phase, making this method nearly unusable for dynamic environments.

(Dual-) Paraboloid Mapping. Another parameterization was proposed by Heidrich and Seidel [Heidrich98b]. Here the analogy is the image obtained by an orthographic camera viewing a perfectly reflecting paraboloid. When compared to sphere or cube mapping, this parameterization introduces less artifacts because the sampling rate only varies by a factor of 4 over the complete hemisphere. For 360°views, two parabolic maps can be attached back to back.

When comparing these different environment mapping techniques, it becomes obvious that the parabolic parameterization would be among the best choices for hemispherical and omnidirectional shadow maps due to the following properties:

- Sampling ratio varies only by a factor of 4.
- One map covers one hemisphere
- Easy to implement (described in Section 6.2)

In the next sections we will describe the concept of paraboloid mapping in detail, first using the mathematical interpretation and later with respect to graphics hardware (implementation).

6.1.1 Theory of Paraboloid Mapping

As described by Heidrich and Seidel [Heidrich98b], the image seen by an orthographic camera facing a reflecting paraboloid

$$f(x,y) = \frac{1}{2} - \frac{1}{2}(x^2+y^2) \qquad ,x^2+y^2 \leq 1, \tag{6.1}$$

contains all information about the hemisphere centered at $(0,0,0)$ and oriented towards the camera $(0,0,1)$. This function is plotted in Figure 6.2(a). Since the paraboloid acts like a lens, all reflected rays originate from the focal point $(0,0,0)$ of the paraboloid.

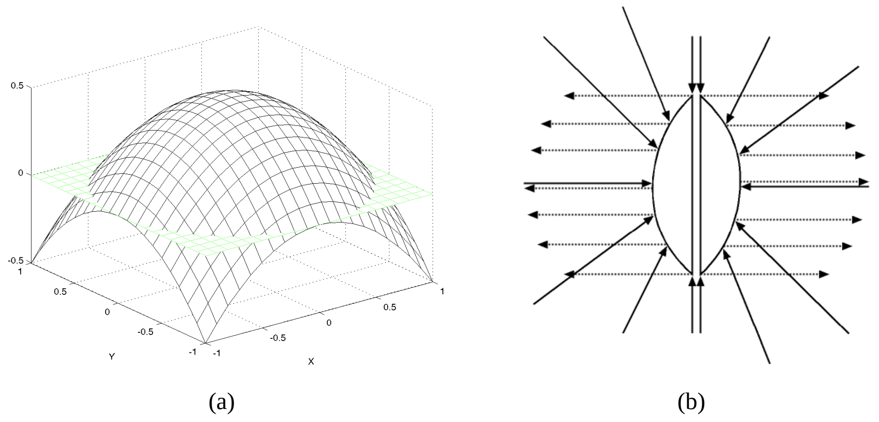


Figure 6.2: (a): Paraboloid $f(x,y)=\frac{1}{2}-\frac{1}{2}(x^2+y^2)$. (b): Using two paraboloids to capture the complete environment.

In order to capture the complete environment (360 °), two paraboloids attached back-to-back can be used, as sketched in Figure 6.2(b). Each paraboloid captures rays from one hemisphere and reflects it to one of the two main directions.

To use the paraboloid as a 3D-to-2D mapping scheme all we have to do is finding the point $P=(x,y,z)$ on the paraboloid that reflects a given direction $\vec{v}$ towards the direction $d_0=(0,0,1)$ (or $d_1=(0,0,-1)$ for the opposite hemisphere). Using Equation 6.1 we find the normal vector at P to be

$$\vec{n}=\frac{1}{z}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad . \tag{6.2}$$

Since the paraboloid is perfectly reflecting we simply calculate the halfway vector $\vec{h}$ which is equal to $\vec{n}$ up to some scaling factor. Using $\vec{h}$ and Equation 6.2 we can now formulate the 2D mapping of $\vec{v}$:

$$\vec{h}=\vec{d_0}+\vec{v}=k\cdot\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad v_z \geq 0. \tag{6.3}$$

For $v_z<0$ d_0 gets replaced by d_1 which corresponds to the other hemisphere (paraboloid).

6.1.2 Dual-Paraboloid Shadow Mapping

As shown in Equation 6.3 the paraboloid mapping can be used to parameterize one hemisphere using 2D coordinates (x,y). These 2D coordinates can of course also be used to perform a shadow map lookup. Instead of computing a perspective projection $(x/z, y/z)$ all we have to do is to use the new mapping instead.

It is clear that this won't work alone because we still need some scalar value representing the depth of a pixel for the actual shadow test. Since the dual approach already divides the environment into positive and negative z regions, we can get along by just using the distance between the given surface point and the center of the paraboloid $(0,0,0)$. This way we extended the original paraboloid mapping to be a 3D-to-3D mapping which retains all information relevant for shadow mapping.

Hemispherical Point Light

In the case of a point light source with a field-of-view of $180°$ one paraboloid map is capable of storing all relevant depth information that can be seen by the light source. The shadow map is generated by first calculating the transformation that translates the light's position to $(0,0,0)$ and rotates the light direction (main axis) into either d_0 or d_1. For all surface points in front of the light source we compute the 2D coordinates (x and y of halfway vector scaled to $z = 1$) and store the point's distance to the origin (light source) at that shadow map position.

During the actual shadow test surface points get first transformed to the new light source coordinate system. If the transformed point is in front of the light source we calculate the paraboloid coordinates and compare the stored depth value with the actual distance of the point to the origin. The point is now in shadow if the stored depth value is less than the actual computed one. Otherwise the point is lit.

Omnidirectional Point Light

For light sources that illuminate the complete environment ($360°$) we have to use the dual-paraboloid approach since one paraboloid map only covers one hemisphere. To generate these two maps we compute the transformation that brings the point light to the origin $(0,0,0)$. For all surface points we first have to check which map is responsible for storing the information. Based on the sign of the z component we choose either the *front-facing* [2] paraboloid with $d_0 = (0,0,1)$ or the *back-facing* one with $d_1 = (0,0,-1)$. After this test the shadow map position and entry is computed as before (using either d_0 or d_1 to compute the halfway vector) and stored in the selected shadow map.

When performing the shadow test all we have to do is to transform the surface point to the light source coordinate system, choose the corresponding shadow map and $d_{0/1}$ based on the sign of the z component and do the shadow map test as in the hemispherical case.

[2]The terms *front-* and *back-facing* are non significant and only used to divide the environment into two regions.

6.2 Implementation

Heidrich and Seidel [Heidrich98b] showed that dual-paraboloid environment mapping can be implemented on standard graphics hardware (e.g. using OpenGL [Segal98]). Since they used a preprocessing step to compute the environment map it is not obvious how paraboloid maps could be generated for fully dynamic environments. One, although very time consuming way would be to first generate a cube map (six rendering passes) and re-sample those images as described in [Blythe99]. Although this could be implemented using graphics hardware it would still be too slow for real time applications.

To speed up this generation phase the obvious way would be to render an image using the paraboloid mapping instead of the perspective projection normally used. Rasterization hardware renders triangles by perspective correct interpolation (homogeneous coordinates). Since this part of the hardware is fixed, we cannot directly map pixels to new positions as the paraboloid mapping would require.

However we can accept this linear interpolation if we assume that the scene geometry is tessellated fine enough. In this case we can simply transform the vertices of triangles according to the paraboloid mapping since the interpolated pixels won't differ too much from the exact solution due to the fine tessellation.

Mapping vertices according to the extended paraboloid mapping (2D coordinates and depth value) can easily be implemented using the so called programmable vertex engines [Lindholm01] available on state-of-the-art graphics cards (see Section 2.3.2). These vertex programs operate on a stream of vertices and replace the former fixed vertex pipeline (transformation, lighting, texture coordinate generation etc.) by a user-defined program. These programs are usually directly evaluated on the graphics hardware resulting in high speed and high flexibility.

6.2.1 Generation of Paraboloid Shadow Maps

Implementing the generation phase for one paraboloid shadow map using vertex programming is straight forward since the programming model supports all operations needed. However we have to be careful about numerical stability. Since we want to include only those pixels that a really part of the hemisphere belonging to the chosen z axis (d_0 or d_1) we have to find a way of culling away unwanted pixels. Theoretically this test could be based solely on the range of valid coordinates $x^2+y^2 \leq 1$ (Equation 6.1). For the implementation this fails due to numerical problems: When using $d_0 = (0,0,1)$ the normalization for points with $z \rightarrow -1$ would break due to the singularity at this point. If we would hand down these *undefined x* and y coordinates to the rasterization engine it would result in an undefined polygon being rasterized. The solution to this is a per-pixel culling test. During generation of the shadow map we calculate an alpha value based on the z coordinate of the transformed vertex. This alpha value is mapped to an unsigned value $[0;1]$ by an offset of 0.5. Using the alpha test we can now cull away pixels based on the sign of the z coordinate which corresponds to either the front- or back-facing hemisphere being rendered.

With this scheme we can now implement a vertex program to generate a paraboloid shadow map for one hemisphere d_0 according to the following pseudo-code[3]:

```
$P' = M_{light} \cdot M_{model} \cdot P$
$P' = P'/P'_w$
$output\ alpha = 0.5 + \frac{P'_z}{z_{scale}}$    /* for alpha test */
$len_{P'} = ||P'||$
$P' = \frac{P'}{len_{P'}}$
$P' = P' + d_0$    /* halfway vector */
$P'_x = \frac{P'_x}{P'_z}$    /* x paraboloid coordinate */
$P'_y = \frac{P'_y}{P'_z}$    /* y paraboloid coordinate */
$P'_z = \frac{len_{P'} - z_{near}}{z_{far} - z_{near}} + z_{bias}$    /* distance */
$P'_w = 1$
$output\ position = P'$
```

First the incoming vertex is transformed and normalized by its w component. Next, an alpha value $\in [0;1]$ is computed by scaling the z component by some user-defined constant (e.g. light's far plane z_{far}) and biasing it by 0.5 to conserve the sign of z. Using an alpha test function $\alpha \geq 0.5$ pixel values belonging to the opposite hemisphere d_1 are culled away.

Finally, we compute the paraboloid coordinates P'_x and P'_y as described previously. For P'_z we assign the scaled and biased distance from P' to $(0,0,0)$. Due to the precision of the depth buffer we have to use appropriate scaling and biasing factors here, similar to the selection of near and far clipping plane when using a perspective projection[4]. To avoid self shadowing artifacts we also have to move the z component slightly away from the light source using a bias value z_{bias} (similar as in [Reeves87]).

Generating a shadow map for the opposite hemisphere d_1 is trivial since we only have to flip the sign of P'_z after the transformation step and use the parameterization of d_0 as before:

```
$P' = M_{light} \cdot M_{model} \cdot P$
$P'_z = -P'_z$
...
```

6.2.2 Shadow Mapping with Paraboloid Shadow Maps

Implementing the shadow test using paraboloid mapping is as trivial as the generation step. In the case of an omnidirectional point light we compute the mappings

[3]Instead of the assembler code we choose a more readable form here.

[4]Computing the distance to the origin means that we have near and far clipping *spheres* instead of planes.

for P'_0 (direction d_0) and P'_1 (direction d_1) and assign these as texture coordinates for texture unit 0 and 1. This step requires an additional scale and bias operation since texture coordinates need to be in the range of $[0;1]$.

In addition to this we also have to compute a value for selecting the right paraboloid map. By computing an alpha value based on the sign of the z component of P'_0 we can later select the appropriate texture map by checking $\alpha \geq 0.5$.

Finally the *normal* OpenGL vertex operations (camera transformation, perspective projection, lighting, additional texture coordinates etc.) are applied.

In pseudo-code the vertex program for rendering from the camera view using two paraboloid shadow maps will look like this:

```
$P'_0 = M_{light} \cdot M_{model} \cdot P$
$output\ alpha = 0.5 + \frac{P'_{0,z}}{z_{scale}}$
...
$P'_0 = P' + d_0$    /* hemisphere $(0,0,1)$ */
...
$texcoords_0 = 0.5 + 0.5 \cdot P'_0$    /* texcoords $\in [0;1]$ */
$P'_1 = M_{light} \cdot M_{model} \cdot P$
...
$P'_1 = P' + d_1$    /* hemisphere $(0,0,-1)$ */
...
$texcoords_1 = 0.5 + 0.5 \cdot P'_1$    /* texcoords $\in [0;1]$ */
normal OpenGL calculation for
lighting and vertex position
```

During the texturing stage we select the appropriate shadow test result depending on the computed alpha value:

```
$if(\alpha \geq 0.5)\ then$
  $res = tex_0$    /* 1 for lit, 0 for shadow */
$else$
  $res = tex_1$    /* 1 for lit, 0 for shadow */
$endif$
$output_{RGB} = res \cdot full\ illumination\ +$
     $(1 - res) \cdot ambient\ illumination$
```

This can be implemented using programmable texture blending (e.g. NVIDIA's register combiners [NVIDIA02]) which is available on all state-of-the-art graphics cards.

For hemispherical point lights only one texture unit is used for shadow mapping and the result of one *if*-clause sets $res = 0$ (one hemisphere always in shadow).

6.3 Results

We implemented and tested our shadow mapping technique on an AMD Athlon Linux PC equipped with a NVIDIA GeForce3 64Mb graphics card using OpenGL as a graphics API. This cards supports vertex programs, programmable texture blending and also multitexturing with up to four texture units. This way we can include the shadow test when rendering the final scene while still having two (or three) texture units left for the scene's textures.

As stated in previous sections, the number of rendering passes is two for hemispherical point lights (one shadow map generation and one final rendering pass) versus three for a omnidirectional light source (two for generating two hemispherical shadow maps and one final rendering pass using two texture units for the shadow test).

Figure 6.3 shows a scene illuminated by an omnidirectional point light source located in the middle of the room. This scene was directly imported from 3D Studio Max and only the wall polygons have been tessellated further (each wall subdivided into 64 quads) to avoid the interpolation problems addressed in Section 6.2. On a GeForce3 this scene can be rendered in real time ($>$ 30 frames per second) including dynamic update of shadow maps at each frame (three rendering passes in total).

Figure 6.4 shows the results for a hemispherical light source located at the top of the room. As in Figure 6.3 only the surrounding walls and the floor polygon had been further subdivided. In this example an additional texturing unit is used to include the scene's textures in the final pass. This scene can also be rendered at real time rates (one shadow map generation pass and one final pass).

Both examples had been rendered using a resolution of 512 by 512 pixels for both, shadow maps and final image. Shadow quality can be further improved by generating high resolution maps using offscreen buffers.

6.4 Discussion

In this chapter we have shown that the dual paraboloid approach presented by [Heidrich98b] can easily be adopted for shadow mapping. This approach is well suited for hemispherical and omnidirectional light sources. By utilizing advanced graphics cards features the algorithm runs completely in hardware and uses only a minimal amount of rendering passes.

Although the algorithm fails for very large polygons due to the linear interpolation performed during rasterization this is in most cases no problem: For games and other interactive applications most geometry is already very high-detailed. The only exception to this are large polygons representing walls, floor, ceiling etc. These polygons could either be further tessellated or simply ignored during the shadow map generation phase. Later is valid in cases where the visible part of the scene is bounded by such polygons, meaning those parts of the scene won't cast visible shadows anyway.

Since our method differs from the traditional shadow mapping approach only in terms of parameterization, we are able to apply all known quality improvements (e.g. percentage closer filtering [Reeves87]) without any efforts.

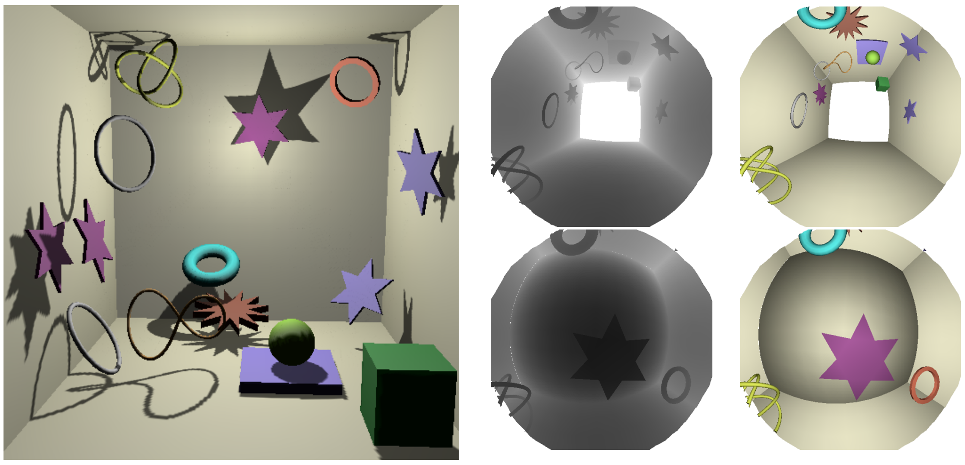


Figure 6.3: Left: Scene illuminated by an omnidirectional point light (located in the middle of the room). Middle column: The two paraboloid shadow maps used (shown as grey scale images). Right column: The two hemispheres as seen by the light source.

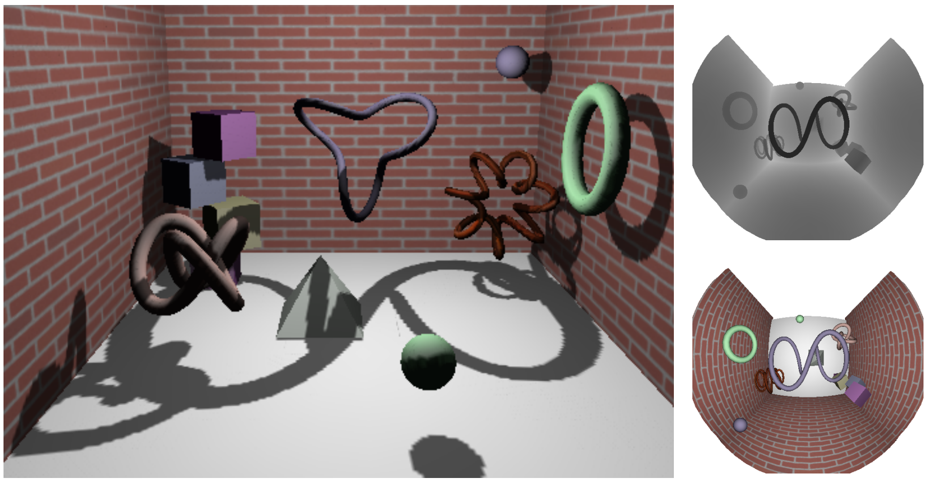


Figure 6.4: Left: Scene illuminated by a hemispherical point light. Right column: Shadow map and light source view (paraboloid mapping).

Chapter 7

Extended Light Maps

Recalling the different shadow map algorithms in the previous chapters it becomes clear that shadow mapping is very well suited for hardware-accelerated rendering. This is especially due to the fact that the shadow map itself can be generated using rasterization hardware, so no additional effort has to be made during this step. Since the shadow map generation phase does only compute the z values of the frontmost pixels, a large number of hardware resources, e.g. texture mapping capabilities, remain unused during this step.

In this chapter we describe a technique that combines the power of light mapping with the traditional shadow mapping algorithm in a very efficient way. Light maps are precomputed textures that are either directly mapped onto the surfaces or projected into the scene, as done by [Segal92]. Light mapping is mostly used to overcome the limitations imposed by poor per-vertex lighting (e.g. to render high quality specular highlights on large polygons) but is also suitable for special effects like simulating slide projectors [Heidrich98a].

We call this combination an *extended light map*, since we add an additional shadow channel to the RGB light map. In the following sections we describe our approach in more detail, give some example applications and finally show some results. As an underlying graphics API we refer to the OpenGL rendering pipeline [Segal98, Woo99] and some of the extensions proposed by [NVIDIA02], but it is also possible to map the algorithm to other graphics APIs such as Microsoft's DirectX [Microsoft00].

7.1 Extended Light Maps

Modern graphics hardware is capable of rendering large polygonal models with several simultaneously applied textures and lights at real-time frame rates. However, as speed and quality increases, it is always important to utilize as much hardware resources as possible in a single rendering pass to achieve good looking results at high frame rates. The idea of rendering with extended light maps is simple but very efficient: To compute shadows using the traditional shadow map approach, one has to render the whole scene as seen by the light source position. Normally this is done by rendering the scene and reading back the depth buffer which is later used as a shadow map texture. In the case of utilizing graphics hardware this approach is far from being efficient: rendering the geometry is expensive and no lighting or texture mapping had been applied at this time, which e.g. means that some ultra-fast T&L (transform and lighting) circuits as well as several texturing units had been wasted.

The solution to this dilemma is simple: during shadow map generation, we also compute parts of the illumination of the frontmost surface and combine the resulting light map with the computed shadow map. This has the advantage that much more surface detail can be achieved during the final rendering pass. While performing the normal shadow mapping, which is done via projective texture mapping, we can also apply our precomputed illumination information to lit pixels (pixels that pass the shadow test) at nearly no additional cost.

An extended light map is basically a two-channel 2D texture image storing information as seen by a point light source. The first channel, which we call the *light channel* consist of RGB triples representing some kind of lighting information of the frontmost surface (as seen from the light source). The second channel, called the *shadow channel*, is used to store the depth (z value) of this frontmost surface. This channel is usually stored in the alpha channel of the extended light map and is later used to determine which pixels are lit and which ones are in shadow.

7.1.1 Shadow Channel

Our shadow mapping implementation is based on the color-coded shadow mapping technique described by [Heidrich99] (see Section 3.2.4). The main idea of this algorithm is to encode depth values in the alpha channel of the framebuffer. During shadow map generation, the whole scene is textured using a 1D texture map representing a linear ramp between 0 and 1 in the alpha channel. With automatic texture coordinate generation enabled, one can compute z values in light source space and map these into the alpha channel. This image is stored away in a 2D texture map and will later be projected into the scene. For the final shadow test the scene is now rendered as seen by camera, but this time with two textures enabled: one is the 2D shadow map from the previous pass (which holds z values of the frontmost pixels as seen by the light source), while the other is the linear ramp 1D texture which now generates the z values in light source space of the frontmost pixels as seen by the camera. Using state-of-the-art graphics hardware, e.g. NVIDIA's register combiners [NVIDIA02], the shadow test can now be performed by subtracting these two values and using a conditional test: a pixel is in shadow, if the camera's z value is greater than the corresponding shadow map entry. Otherwise the pixel can be seen by the light source and will be lit.

Shadow Mapping using the Fog Computation

Since our main goal is to utilize as much hardware resources as possible during the shadow map generation pass, we use a slightly modified version of the alpha based shadow map algorithm. Instead of using a 1D texture one could also use OpenGL's fog computation in order to map z values to alpha values. Fog is normally used to fade away objects as the distance to the eye increases. To do this, OpenGL computes the so called *fog factor*. Beside some exponential functions, the fog factor can also be configured to decrease linearly as the distance to the eye increases [1]. In this mode, the fog factor is computed as

$$f = \frac{end - |z|}{end - start} \quad ,$$

where end and $start$ are user defined scalar values. Setting $start$ and end to the light source view's near and far clipping plane, f gives us the same linear mapping as the 1D alpha texture (beside the fact that the mapping is now from 1 to 0 as the distance increases). To avoid numerical problems we also slightly offset (bias) the z values so that lit pixels don't shadow themselves in the final rendering pass. The bias is applied by simply subtracting a small positive value ($bias_z$) from both, the end and $start$ value. So the fog factor will be computed as

$$f = \frac{light_{far} - bias_z - |z|}{light_{far} - light_{near}} \quad .$$

Instead of blending incoming fragments with this fog factor, we directly store f in the alpha channel. Using this method, we don't need any texture mapping units during the generation of the alpha shadow map.

It should be pointed out that this fog based approach can only be applied during the shadow map generation pass. This is due to the fact, that fog is calculated in *eye space*. Since we need z values in the light source coordinate system we still have to use the 1D mapping in the final pass, because there is no way to transform z values before the fog factor is computed. A possible extension for future graphics hardware would be some kind of user-defined reference plane for fog computation, so that we can perform shadow mapping without any "helper" textures.

7.1.2 Light Channel

Precomputing Illumination

One possible use of extended light maps is the precomputation of illumination. Having z values encoded in the alpha channel, we still have the *RGB* channels left to store the result of the lighting and texturing computation. Using only ambient and diffuse illumination, we could simply enable the light source, store the resulting color values and map them onto the scene in the final pass. In the case of

[1]Note that the distance is computed as the distance to the eye plane $(0,0,1,0)$.

specular reflections this would be incorrect, because the computation of specular highlights is done using the so called *halfway vector*, which is the normalized vector between viewing and light direction. These would always be the same since we are rendering the scene as seen by the light source. The idea is to separate the specular part from the calculation of ambient and diffuse illumination. To get the correct halfway vector we simply use an additional light source for the specular contribution which is placed at the final camera position. With this setup we have just swapped the meaning of ”viewing direction” and ”light direction” during the light map generation step , but this swap operation has no effect on the halfway vector anyway.

What is now incorrect is the computation of the spotlight effect and attenuation. The spotlight effect, which describes some kind of intensity fall off with respect to the main spotlight direction, would now be computed using the camera’s direction. There are two solution to this problem. One would be to simply ignore the spotlight effect and have the additional light source act like a point light source, while the other solution involves some kind of *spotlight texture*, as described by [Segal92]. The later has the benefit, that spotlight textures are applied on a per-pixel basis and therefore usually look much more convincing then the per-vertex spotlight effect computed by OpenGL. If a spotlight texture is used, it should be applied to both light sources, not only to the specular one.

To solve the problem of correct attenuation one could also use some kind of 1D texture map, representing the combination of constant, linear, and quadratic fall off, but this would probably be a waste of resources. A cheap approximation could be achieved by moving the specular light source along the camera direction so that the distance to objects is nearly equal to the distance as seen by the light source position.

Environment Mapping

The light channel could also be used to map all kinds of reflections onto the frontmost surfaces. OpenGL supports various methods for computing texture coordinates suitable for environment mapping. Nearly all of the them are calculated using the reflection vector

$$\mathbf{r} = \mathbf{u} - \mathbf{2nn^T u}$$

which is based on the vector $\mathbf{u}$ pointing from the origin to the vertex and the current normal vector $\mathbf{n}$. If we want to use these texture coordinates during the light channel generation phase, we have to ensure that the reflection vectors are calculated in the camera’s eye space, not in light source space. We can achieve this using a matrix setup that is normally a ”don’t do” in OpenGL programming. For the actual viewing matrix, we use the camera viewing matrix. This yields correct eye space coordinates during the texture generation phase. For the projection matrix, we concatenate the light’s projection and viewing matrix and also multiply the inverse of the camera's viewing matrix to the right. So the setup is something like

$$\begin{aligned} \mathit{Viewing} &= \mathbf{C_{view}} \\ \mathit{Projection} &= \mathbf{L_{proj}L_{view}C_{view}^{-1}} \quad . \end{aligned}$$

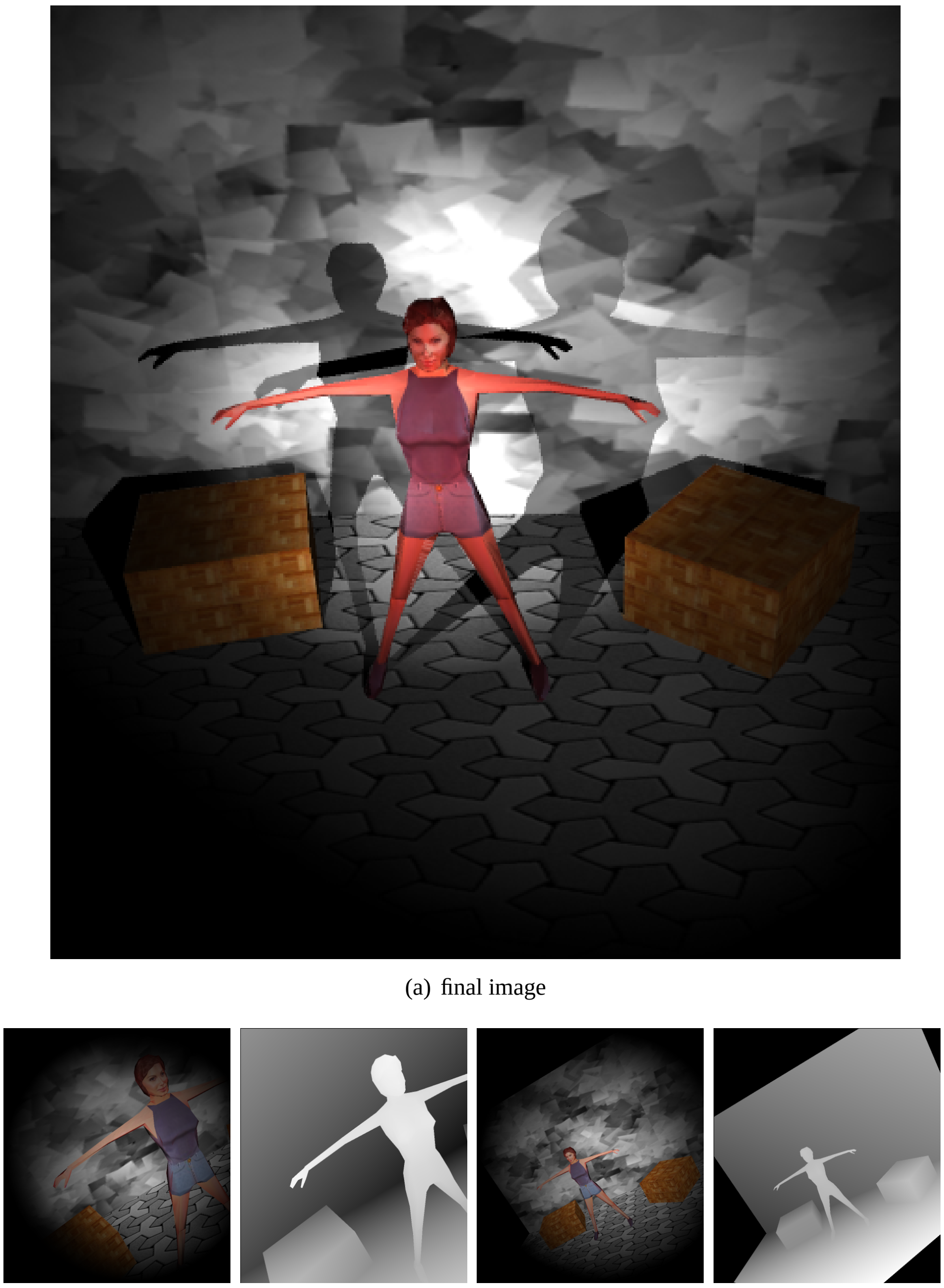
This enables us to render the scene from the light source view, but with all eye space computations done in the camera's eye space, which is important for proper calculation of texture coordinates used for environment mapping. One drawback of this approach is that also the fog computation or the 1D ramp texture used for shadow testing will no longer be performed in light source space. So we have to be careful that the coordinate system used during the generation of the shadow channel is somehow compatible to the one used in the final shadow test phase.

7.2 Results

We have implemented the approaches described in this chapter using a standard PC with a NVIDIA GeForce2 GTS 32MB graphics card. This card is able to perform up to two texturing steps simultaneously and does also support the NVIDIA register combiner extension [NVIDIA02].

Figure 7.1 shows the result of using two extended light maps to illuminate a given scene. During the generation phase, lighting and texturing is computed and stored away in the light channel of the extended light map. In the final rendering pass, only geometry with no additional lighting or texturing is rendered, so every surface detail is part of the extended light maps. Even at very high resolution (1024×1024 pixels for the extended light maps) the scene can still be rendered at real-time frame rates on a NVIDIA GeForce2 GTS. Rendering this scene requires four passes in total (two for the generation of the extended light maps plus two for the final image).

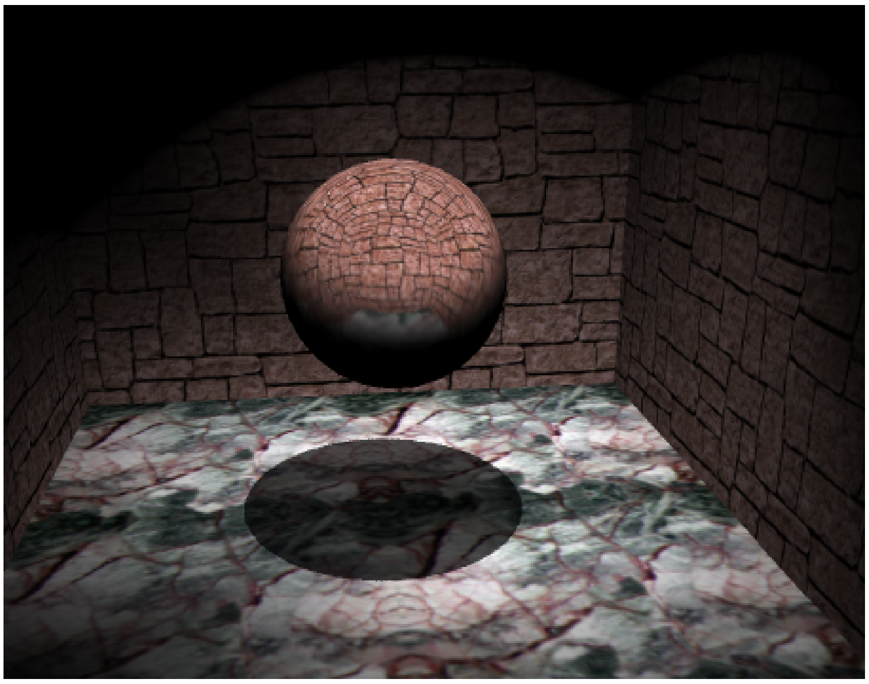
In Figure 7.2 the light channel is used to perform some kind of environment mapping for the sphere in the center of the room. This scene uses three passes in total, one for the generation of the extended light map, one for normal lighting and object textures plus one final pass to apply the extended light map. Reflections are rendered using a precomputed cube map [NVIDIA99] where the six faces correspond to the wall, ceiling and floor textures.



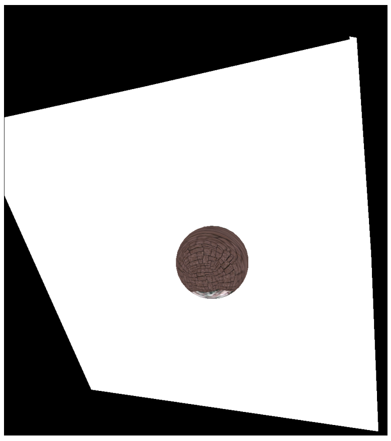
(a) final image

(b) light ch. 1 (c) shadow ch. 1 (d) light ch. 2 (e) shadow ch. 2

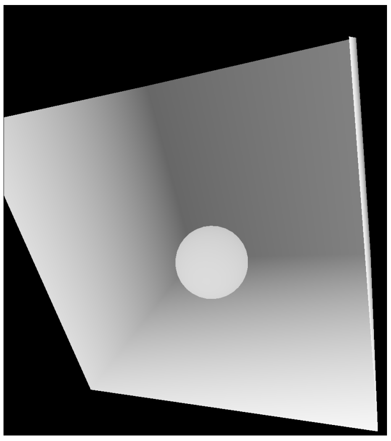
Figure 7.1: A scene illuminated by two extended light maps.



(a) final image



(b) light channel



(c) shadow channel

Figure 7.2: A simple scene with one extended light map used for environment mapping.

7.3 Discussion

In this chapter we presented a way of combining shadow maps with light maps. The benefit of this approach is that additional hardware resources are available during the shadow mapping step. These additional resources can be used to achieve better image quality while reducing the number of rendering passes normally needed for these kind of effects. It has to be pointed out that this precomputed light channel will only be applied to lit pixels, since information is only available for pixels seen by the light source. So in the case of environment mapping it is not possible to have reflections appear in shadowed regions. Another problem related to accurate shadowing is the resolution and depth of the framebuffer. In our implementation the maximal size of an extended light map was 1024×1024 pixels. This is enough for most applications but in order to achieve high quality images it would be necessary to use high resolution offscreen buffers or the ability to render directly into textures, which is currently only possible on some SGI workstations and on a few PC graphics cards. The framebuffer's depth is also important to achieve reasonable shadow quality for highly detailed scenes. Most of the graphics hardware available today is still using 8 bits for each color channel. This results in a very poor sampling quality of depth values. Using floating point offscreen buffers available on some newer PC-cards color and depth precision is no longer an issue.

Part II

Advanced Shadow Volumes Techniques for Point Light Sources

Chapter 8

Shadow Volumes for Interactive Global Illumination

While the previous part of this thesis was dedicated to the shadow mapping technique, we will now switch to the second *main* shadow method that is used in hardware-accelerated rendering today: shadow volumes. In contrast to the image-based shadow mapping technique, the shadow volume method performs all calculations in object-space, resulting in very precise shadowing information.

In the context of high-quality rendering, e.g. in architectural applications, very precise lighting calculations are needed in order to obtain physically meaningful results. This can be achieved by using global illumination techniques that approximate the global light transport within an environment. Without going into detail, it is obvious that a major part of any global illumination technique is the computation of visibility.

In this chapter we will show how hardware-accelerated shadow volumes can be combined with a global illumination system. In our hybrid rendering system, we choose to compute the direct illumination (including shadows) using graphics hardware, whereas the more complicated indirect illumination is calculated using software rendering. Since our system is dedicated to the rendering of very complex scenes, we support only point light sources where all direct illumination can be computed in hardware. More complex light source, such as area lights, can of course be computed using the indirect illumination system.

One main aspect when implementing a rendering system for complex scenes, which usually consists of many light sources, is the efficient rendering of shadows. Rather than iterating over individual light sources and accumulating contributions our implementation precomputes screen-spaced shadow masks that capture shadow information for groups of light sources, thereby reducing the number of rendering passes needed. These shadow masks not only store simple boolean visibility (lit or shadowed) but also quantitative information about the light source's energy. This way we can easily integrate non-uniform energy distributions, e.g. given in the form of goniometric diagrams, into a hardware renderer.

8.1 Motivation

In order to compute the energy arriving at a given surface, the first task is to determine the sources of energy, which may be direct sources, such as lights, or indirect sources, such as reflected light. Given these sources of energy, we then have to check if energy originating from a given source hits the surface or is blocked by objects inbetween.

Although we already discussed this shadow problem in the previous chapters, the computation of shadows in a global illumination system is of special interest. Previously, we only considered the light sources as the only source of energy and therefore concentrated on methods that efficiently compute shadows for a rather limited number of lights. In the case of indirect illumination, these techniques fail due to the overhead of building up the necessary data structures, e.g. in the case of shadow maps. Many global illumination systems therefore use geometric methods or shadow rays to compute visibility.

Due to the enormous numerical calculations involved in global illumination computations, energy transport (including visibility) is usually computed as an average value for patches of the scene. Patches are single elements of the scene's mesh structure, e.g. triangles or quadrilaterals. Computing a global illumination solution is therefore directly related to the number of patches in the scene.

Neglecting shadowing, a very course tessellation would be sufficient, since direct and indirect light usually does not contain sharp intensity changes. Adding shadowing, the scene's mesh structure is directly related to the shadow quality. Since blocking geometry introduces high-frequency intensity changes, one can only achieve accurate shadow boundaries by a very fine mesh tessellation or a mesh refinement strategy that generate patches with edges aligned to the shadow boundaries, a method known as discontinuity meshing [Heckbert92].

As seen in the previous part of this thesis, shadows in interactive, dynamic environments can be generated very efficiently using the shadow map technique. However, due to the image-based nature of this method, sampling artifacts are likely to occur, even when advanced filtering techniques and better parameterizations are used or the resolution of the shadow map is increased.

In Section 3.2.5 we introduced Crow's shadow volumes [Crow77] as one of the most popular object-space shadow methods that can be efficiently mapped to graphics hardware using the stencil-counting scheme proposed by Heidmann [Heidmann91]. The main benefit of using shadow volumes is that this approach generates pixel correct shadow boundaries, which is very important for high-quality rendering.

Since our focus is on the direct illumination part of the system, especially in the shadow part, we will only briefly describe the indirect illumination method. A more detailed description of the whole system can be found in [Dmitriev02].

8.2 Global Illumination

The global illumination part of our hybrid rendering system is based on Quasi-Monte Carlo photon tracing and density estimation techniques.

Our system exploits temporal coherence of illumination by tracing photons selectively to the scene regions that require illumination update. Such regions are identified by a small number of so called pilot photons. Based on the pilot photons which require updating, we detect photons with similar paths due to periodicity in the multi-dimensional Halton sequence, which is used to generate photons.

If not all invalid photons can be updated during a single frame, frames are progressively refined in subsequent cycles. The order in which the photons are updated is decided by inexpensive energy- and perception-based criteria whose goal is to minimize the perceivability of outdated illumination.

8.3 Direct Illumination

For computing the effect of direct illumination we chose to use OpenGL-compliant graphics hardware. Instead of restricting ourselves to the fixed function pipeline of standard OpenGL we utilize programmable vertex and pixel hardware to gain the highest qualitative and most efficient rendering possible. Our current implementation is customized to run on NVIDIA GeForce3 graphics cards, but can of course also be implemented on any other card with similar features (e.g. ATI's Radeon series).

One main aspect of image quality is the accurate representation of shadows. Although shadow mapping[Williams78] is directly supported by the graphics hardware it is problematic due to its sampling problems. Another disadvantage is that for dynamic environments shadow maps change very often which would lead to a huge amount of regeneration passes. We therefore prefer the shadow volume algorithm proposed by Crow[Crow77] since it

- generates very precise shadows by performing calculations in object space,
- can be efficiently implemented using graphics hardware, and
- is appropriate for dynamic environments.

For complex scenes the generation of shadow volumes, which requires finding the silhouette edges of all objects with respect to the light source, is quite expensive. By exploiting temporal coherency of shadow volumes we can limit the regeneration of shadow volumes to those objects that are moving and reuse the volumes of all static objects from the previous frame.

Our shadow volume implementation is based on the hardware stencil buffer scheme presented by Everitt et al. [Everitt02], which solves the problematic cases of shadow volumes intersecting the near clipping plane.

With normal OpenGL the shadow volume algorithm would require $N+1$ rendering passes where N is the number of shadow casting light sources. However using programmable features available on recent graphics hardware we are able to collapse up to 4 passes into a single one. This is done by first rendering the scene's geometry as seen by the camera, resulting in having the depth values of the front most pixels in the depth buffer. After this we loop over the first four light sources where in each step we first initialize the stencil buffer and draw shadow volumes with the corresponding stencil operation (increment/decrement). The content of the stencil buffer, which corresponds to the shadow test result, is then copied to one of the color buffer channels (red channel for first light, green channel for second, etc.) and the stencil buffer is initialized for the next light source. After this loop we obtain a RGBA *shadow mask* containing the shadow information for up to four light sources.

In the final rendering pass we then render the scene once again but this time using a customized vertex program [Lindholm01] which instead of summing up all lighting contributions for a given vertex uses additional output attributes that hands out the illumination for light source $L_{0\ldots3}$ separately. These values will then be linearly interpolated over the primitive (triangle) and passed to the texture blending stage.

Having the shadow mask as an projective RGBA texture we can apply the shadow result separately for each of the four light sources and output the total illumination as the pixel value.

$$out = indirect + illum(L_0) \cdot mask_R + illum(L_1) \cdot mask_G + \ldots$$

For more light sources this approach can be extended by simple multipass rendering. So for N light sources we have to generate $\lceil N/4 \rceil$ shadow masks. Another $\lceil N/4 \rceil$ passes are needed to compute the illumination. These passes can then be summed up using additive blending or the accumulation buffer.

This shadow mask scheme is not only very efficient but also enhances image quality. Using the normal loop scheme, the contributions of all light sources need to be summed up using either the accumulation buffer or additive blending. Since both operations are performed at the end of the pixel pipeline precision is limited to (normally) 8 bits per color channel, whereas our approach sums up at an earlier stage in the pipeline where precision is much higher. Accuracy is also increased for $N > 4$ since we use the accumulation buffer (or additive blending) for groups of four light sources rather than summing up the contribution of individual lights. Another advantage is that our method does not suffer from z-fighting artifacts which normally occur when rendering the scene several times.

8.3.1 Goniometric Diagrams

In order to achieve realistic image quality we also choose to support complex point light sources with non-uniform directional power distribution. Description of these goniometric diagrams are available in standardized formats and are essential for accurate lighting computations.

We include these distributions by re-sampling from the standard format to a cube map texture [Voorhies94], which can efficiently store the complete 360^o view of a point light source and which is supported by the graphics hardware. Figure 8.1 shows an example of a goniometric diagram represented as a cube map texture.

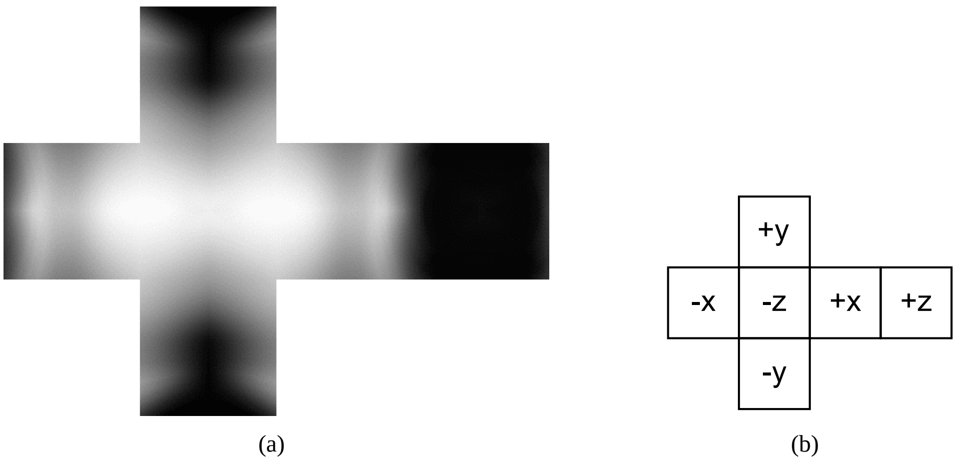


Figure 8.1: Goniometric Diagram.

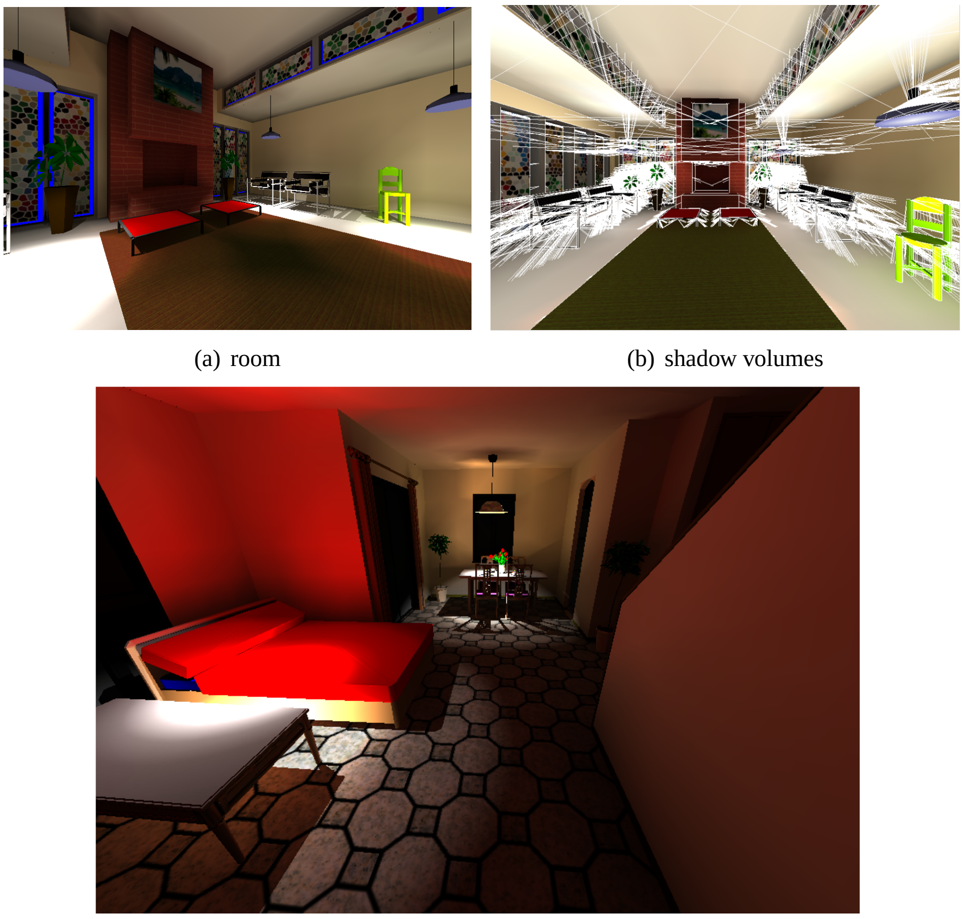
Restricting ourselves to monochromatic goniometric diagrams we can include these as an additional scaling factor for the local illumination of a given light source. Referring to Equation 8.1 the *mask* scaling factor, which was considered to be either 1.0 for lit and 0.0 for shadowed pixels, can be used to perform this additional scaling. In the first rendering pass we render using the appropriate cube map textures and store the scaling factors in one of the color channels. Using four texture units and RGBA masking we are able to generate these for 4 light sources simultaneously. When copying the result of the shadow test back to the color buffer we set all pixels to 0.0 where the shadow test succeeded. Although the dynamic range of these textures is limited to 0.0 to 1.0 this method still improves image quality significantly while introducing only a minimal overhead. Using textures with floating point precision, available on very recent graphics boards, the full dynamic range of the goniometric diagrams is available. But this precision does not come for free, since floating point textures consume up to four times more memory.

8.4 Results

Figure 8.2 shows some example images obtained from our hybrid rendering system running on 1.7 GHz Dual P4 Xeon processors, and an NVIDIA GeForce3 64 MB video card.

Figure 8.2(a) consists of about 12.400 mesh elements and is illuminated by four light sources with goniometric diagrams. Figure 8.2(b) illustrates the shadows volume boundaries generated. For this scene we obtain frame rates of about 8 fps, which includes the display of indirect lighting and direct lighting with shadow computations performed by the graphics hardware. Our current implementation does not support triangle strip generation which affects the refresh rate figures, however, this is not a limiting factor for the performance of indirect lighting display which is refined at an even slower pace.

A more complex scene is shown in Figure 8.2(c). Here the scene consists of about 377.400 mesh elements and two light sources with goniometric diagrams. An interactive session using this scene runs at about 1 fps.


(a) room

(b) shadow volumes

(c) house

Figure 8.2: Interactive session snapshots.

8.5 Discussion

In this chapter we presented a hybrid rendering system for interactive rendering of complex environments, such as architectural scenes. We showed how shadow volumes can be used efficiently for a number of light sources by storing the visibility information for groups of light sources in so called shadow mask textures rather than iterating and accumulating individual light source contributions. One main aspect of our shadow volume implementation is the support of non-uniform light source distributions. By including these quantitative information into the shadow masks we can improve visual quality of the rendered images without major performance loss.

For future work we would like to further improve the efficiency of our OpenGL renderer. This can e.g. be achieved by using connected primitives, such as triangle strips, rather than individual mesh elements. Also, for more complex scenes, hardware-based occlusion culling mechanisms will definitely help to improve rendering speed.

Concerning the shadow volume implementation, there are a number of improvements possible. First of all, silhouette generation is still performed on the host processor, consuming computation resources which could otherwise be used by the indirect illumination part of the renderer. In Chapter 9 we will present a fully hardware-accelerated shadow volume implementation (including silhouette detection), that could solve this problem.

Chapter 9

Shadow Volumes on Programmable Graphics Hardware

In the previous chapter we have shown that stencil-based shadow volumes can be used to generate precise, high quality shadows. This is due to the fact that shadow information is generated in object space, meaning that shadow information is available for every window-space pixel. Achieving information that precise is hardly possible with a sampling based method like e.g. the different shadow map approaches we have seen in the first part.

But this accurate shadow information does not come for free. Generating the necessary silhouette information can put a heavy load on the CPU, and rendering the extruded shadow volumes on graphics hardware can easily exhaust fill rate capabilities.

Another problem is the hybrid nature of this algorithm. In order to produce accurate shadow volumes, both, the CPU and the graphics hardware have to be synchronized. This not only refers to the data transfer, but also, most importantly, to a consistent numerical precision during all calculations.

Nowadays this *perfect* synchronization becomes even more important. Recent graphics hardware exposes powerful programming features that allow nearly arbitrary operations on both vertex and pixel data. When using these programmable features in conjunction with shadow volumes the silhouette extraction performed on the CPU becomes even more problematic.

In this chapter we address these issues and present a method for implementing the whole algorithm on the graphics hardware. Migrating the silhouette extraction to graphics hardware solves a number of issues:

- All calculations are performed on the same hardware, resulting in consistent precision.

- Shadowing objects can be used with programmable vertex processing in the same manner as with fixed transformation processing.
- Applications gain more CPU time, since resources which were formerly dedicated to silhouette extraction are released.
- Rendering with shadow volumes no longer needs to be synchronized with CPU and graphics hardware, minimizing potential idle time on both processing units.
- Silhouette extraction is performed on large chunks of data in parallel (bulk processing), which results in enormous speed up.
- Using the shadow volume approach in an application becomes very simple since only trivial, local preprocessing of the objects needs to be performed. This is especially important for *general* scene graphs, where silhouette extraction would require traversal of all possible transformation and deformation nodes.

9.1 Motivation

Recalling the different steps of the stencil-based shadow volume algorithm (see Section 3.2.5), it becomes clear that silhouette detection and shadow volume extrusion are the only steps that still have to be performed on the CPU. This can not only become a bottleneck if shadow casting objects are highly tessellated, but is indeed problematic if the input geometry will be deformed by the graphics hardware. Current state-of-the-art graphics boards provide powerful, programmable vertex processing units (vertex programs) [Lindholm01] which can be used for nearly arbitrary geometric transformations, e.g. displacement mapping or matrix palette skinning. Using vertex programs in conjunction with shadow volumes requires the CPU to emulate all vertex processing in the same way as it is actually done by the graphics hardware.

As a consequence, detecting silhouette edges is no longer a trivial and fast operation since vertices and face normals have to be re-calculated at every frame in the worst case. Also numerical differences can lead to strange artifacts, e.g. light leaks. As an example, the result of calculating $sqrt(x)$ can significantly differ since CPU and graphics hardware may use different approximations.

Another bottleneck when using the common hybrid approach is that CPU and graphics hardware need to be synchronized such that all shadow volumes are generated when the graphics hardware is ready to render them. Especially in applications like games the CPU is more and more dedicated to handle input events, artificial intelligence or sound, and all graphic-related work should ideally be done by the graphics hardware. Therefore keeping these two processing units asynchronous reduces potential idle time on both.

In the following sections we will show how silhouette detection and shadow volume generation can be implemented on programmable graphics hardware, which solves the described problems.

9.2 Silhouette Detection

In the first step of our hardware implementation we need to bring the actual geometry and the light sources into a common coordinate system. We will choose to transform both to world space, which is view-independent and also, with respect to the scene geometry, reusable for different light sources.

Since all graphics cards perform a combined transformation which also includes the viewing transformation, we begin by setting the viewing matrix to identity. This way every vertex is transformed to world space. Secondly, every vertex in the scene is assigned a unique index number, so that it can later be referenced by its index. For objects which are referenced multiple times, e.g. an object placed in the scene at different locations, we need to ensure that vertices with different transformations also obtain different indices.

With these vertex indices we are now able to dump the world space coordinates of each vertex to the graphics hardware. The result of this step is a texture, in which each texel stores the world space positions of one vertex. The mapping of a vertex to its position in the texture is defined by the vertex index. To retain as much precision for the vertex positions as possible, we use a 4-channel (RGBA) offscreen buffer with floating point precision as output buffer.

We will now explain how this step can be implemented with the help of a vertex program: Instead of rendering filled primitives, like triangles or quadrilaterals, only the vertex itself is rendered as a point. We use the vertex program to compute the position (x,y) in the output buffer from the vertex index (passed along as a vertex attribute) and specify the result as the output position for the vertex. This way each vertex gets rendered as a single pixel at the position (x,y). The final task now is to set the color at position (x,y) to the corresponding vertex's world space coordinates. Since the vertex color output of a vertex program gets clamped to $[0 \ldots 1]$, we output this value using one of the unclamped output registers, e.g. one of the 4D texture coordinates, and then map it to the color register in a fragment program. Note that the vertex's world coordinates include all vertex transformations (i.e. modeling or procedural transformations). This step is graphically explained in Figure 9.1 (Step 1).

The next task in our algorithm is the classification of possible silhouette edges. Assuming that all meshes used in the scene are well-modeled (2-manifold), meaning that there are no open edges and every edge connects exactly two triangles, the silhouette test only needs four vertices and the light source position(s) as input data. Two vertices are used to locate the edge itself whereas the remaining two are used to construct the two triangles that meet at the given edge.

As explained in Section 3.2.5, the silhouette test consists of checking the front-

and back-face condition of the two triangles with respect to the given light source.

Our implementation is therefore straightforward: Given the connectivity information (edges) for all meshes we also assign all edges an unique identifier (index), used for later referencing. Since connectivity and index numbers remain constant, this can be implemented as a preprocessing step.

To detect silhouette edges, we use a brute-force approach that tests every edge in every frame during runtime (except for simple cases, where scene and light sources remain constant). Doing this on the host processor can be quite expensive, but on the graphics card, which can be seen like a SIMD-like (single instruction, multiple data) processor, this is an efficient operation running in parallel.

Like in the world space transformation step, we render all edges as single points. As before, these points represent the index number of a given edge, and have no real geometric meaning. Along with the index number describing the position where to store the result of the edge computation, we also pass all relevant input data for the given edge as additional per-vertex (point) attributes. As stated before, this input data consists of a total of four indices referring to the points that make up the two adjacent triangles. Since the light source position remains constant for all edges, this parameter is set globally.

Testing if a given edge is a silhouette edge is now trivial: We bind the dumped vertex positions as a 4-component floating point texture map and use the four indices to get the world space coordinates of all points. Since texture lookups are only possible in the fragment (pixel) processing step, this has to be implemented as a so called fragment shader.

For both triangles that meet at the given edge we calculate the plane equations and compute the signed distance to the light source position. If the signs of the two distances differ, the edge is marked as a silhouette edge. Since the edge's vertex ordering has to be preserved for later steps, we also compute a flag indicating whether the vertex ordering of the front facing triangle corresponds to the order of the edge's vertices.

The result of the silhouette detection, which are two binary flags, are then written to the framebuffer as a color-coded value at the edge-index position.

9.3 Generation of Shadow Volumes

Generating and rendering the shadow volumes using the results of the previous steps is now straightforward.

In a preprocessing step we generate quadrilaterals for all edges, but instead of using the object's vertex coordinates and transformations, each vertex has a total of three indices and one flag:

- Two indices referring to the world space position of the edge's two end points.

- One index referring to the silhouette flag and the vertex-ordering for the given edge.
- A flag (yes/no) indicating whether the vertex should lie on the edge or should be extruded to infinity. Each quadrilateral has two points on the edge and two points that have to be moved to infinity.

Since we only want to render quadrilaterals for silhouette edges, the silhouette flag is used as a trivial reject. If the edge is not a silhouette edge, we move all vertices outside the viewing frustum, e.g. behind the viewer, so that the complete primitive is clipped away. For silhouette edges we either directly output the world space position, or, if the extrusion flag is true, we move the vertex to infinity with respect to the light source direction. Choosing one of the edge's vertices is based on the vertex-ordering flag, which preserves a consistent winding order.

All these steps are implemented as a vertex program and therefore are fully hardware-accelerated. The generated shadow volumes are similar to those generated on the CPU and can now be used for the stencil-based counting scheme. Figure 9.1 illustrates the different steps of the hardware-based shadow volumes algorithm.

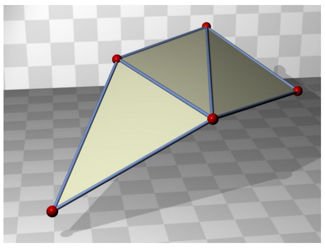
9.4 Implementation

We implemented the described algorithm on an ATI Radeon 9700 graphics card using OpenGL. This card supports all the programmable feature, like vertex and fragment programs with floating point precision, as well as floating point offscreen buffers and textures.

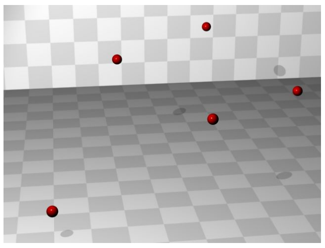
For the first step, we use a floating point RGBA offscreen buffer and modify all of the scene's shaders (vertex programs) such that instead of using the original vertex position the index number is used to calculate (x,y) pixel coordinates and the world space position is written out to the framebuffer. Currently this is a manual step but can simply be automated by a script that generates the modified shaders. Since only the world space position is relevant during this step, we can further optimize it by analyzing which computations inside the shader affect the position and remove all other operations that are only relevant for e.g. the output color or texture coordinates.

For computing the silhouette and vertex-ordering flags, an offscreen buffer with less precision (e.g. RGB with eight bits per channel) is sufficient. By rendering all edges as points and using the offscreen buffer of the previous step as a 2D texture map, we can obtain the four world space position by sampling this texture at the exact integer positions (vertex indices). The light source position is specified as a global parameter. The fragment shader then computes the plane equation and tests for the front-/back-facing condition and the vertex ordering flag. In the previous sections we only discussed the silhouette detection for one light source. However, the fragment shader can easily compute the flags for several light sources

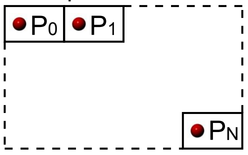
Input: original mesh



Step 1: transform vertices to world space

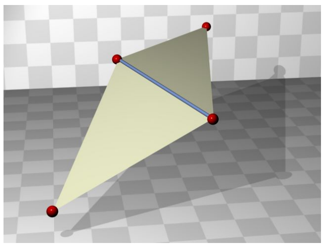


Store position at vertex index (render-to-texture)



Step 2: process edges

Two indices for edge, two indices for adjacent triangles



Get world space positions from Step 1 texture and check front-face / back-face condition.
Store result at edge index (offscreen buffer)

Step 3: render shadow volume quads

Use offscreen buffer from Step 2 as vertex attribute array (silhouette flag) and Step 1 buffer as vertex position array.

```
for each edge {
  if silhouette flag is true
      extrude
   else
      move quad outside view
}
```

Figure 9.1: Workflow for hardware-based shadow volumes.

simultaneously. The number of light sources that can be checked during this step is only limited by the maximum instruction length of the fragment shader and the number of bits available in the offscreen buffer. Since the results for one light source needs two bits, a standard RGB buffer can store the results for up to 12 light sources.

Rendering the shadow volume quadrilaterals requires the results of the two previous steps as input. Since the vertex extrusion and the trivial reject has to be performed before rasterization, this has to be implemented as a vertex program. Unfortunately, there is currently no fast path to access the required data in the vertex program directly. For our algorithm we therefore would propose one of the following features:

- Texture access during vertex processing. First attempts in this direction are made with the release of DirectX 9's displacement mapping, but a more general lookup would be necessary for our method.

- A fast, on the card copy from framebuffer to a vertex attribute array. This could be implemented as a simple copy operation, or ideally as *copy-by-reference* similar to the *render-to-texture* functionality.

Due to the lack of the proposed features, we are forced to use a very slow mechanism that transfers the data from the two buffers to host (CPU) memory and immediately downloads the same data as vertex attribute data for our current implementation.

9.5 Results

Figure 9.2 shows two example scenes with shadow volumes generated using our hardware approach. For both scenes, three light sources are used and silhouette edges are detected for all lights simultaneously. In Figure 9.2 (a) the vertex texture has a size of 128×128 pixels, needed to store the world space positions of the 9326 vertices. The edge buffer has a size of 256×128 which is enough to store the silhouette flags for the 27627 edges. Figure 9.2 (b) has a vertex texture of size 64×64 pixels (9326 vertices) and an edge buffer of size 128×128. The scenes were rendered at a window resolution of 512×512 on an AMD Athlon 1GHz machine equipped with an ATI Radeon 9700 card. For both scenes we obtain frame rates about 20 fps. The main bottleneck here are the framebuffer read backs. With all steps running on the hardware (as proposed in Section 9.4) we expect our method to run considerably faster. Our current implementation should therefore be seen as a proof-of-concept.

Figure 9.3 (left) shows a more complex example illuminated by three light sources. Here the geometry of each of the three spheres is displaced by a procedural noise shader, implemented as a vertex program. Detecting silhouette edges for this scene on the CPU would be very difficult since the vertex program would need to be evaluated on the CPU in order to obtain world space coordinates. Detecting silhouette edges with our hardware method is as simple as for the previous scenes. Only small modifications to the noise shader were necessary which ensure that for each vertex the world space position is passed as a result and the index becomes the vertex's pixel position. Here the vertex texture has a resolution of 64×32 (1638 vertices) and the edge buffer has a resolution of 128×64 (4896 edges).

Since the procedural noise shader only computes new vertex positions, the vertex normals no longer correspond to the actual geometry. Therefore the shading of the three objects looks unrealistic. To avoid strange artifacts we also disabled self and global shadowing for the three objects. With proper shading normals there would be a smooth intensity transition into the shadow region.

Figure 9.3 (right) shows the silhouette edges detected for one light source (yellow) as well as the generated shadow volumes for all three lights (red).

9.6 Discussion

In this chapter we have shown how to perform the silhouette detection step of the shadow volume algorithm in hardware. The benefits of this approach are not only the gain in speed, what is most important is that shadow volumes can now easily be generated for geometry that is transformed by programmable vertex engines, as shown in the procedural noise example. The algorithm itself relies on capabilities available on recent graphics cards: programmable vertex and fragment units, floating point buffers, as well as floating point textures.

An important feature which is currently missing is a fast way to use the contents of a buffer as input for a vertex program, needed when rendering the shadow volumes. We are confident that future drivers will provide a more general memory management functionality. First efforts in this direction are already visible with the upcoming OpenGL 2.0 specification or the so called OpenGL *super buffers*[1].

Also, we did not address the problem of shadow volumes that intersect the near or far clipping plane. A solution to this was presented by Everitt et al. [Everitt02]. As future work we would like to investigate how those special cases can be detected and efficiently processed using our algorithm.

Another possible application for the silhouette detection presented here is in the context of non-photorealistic rendering (NPR). Here the silhouette information could be used to achieve toon-like or pencil drawn shading effects on a per triangle basis, rather than using image-spaced techniques.

[1]See `www.opengl.org` for more details on these specifications.

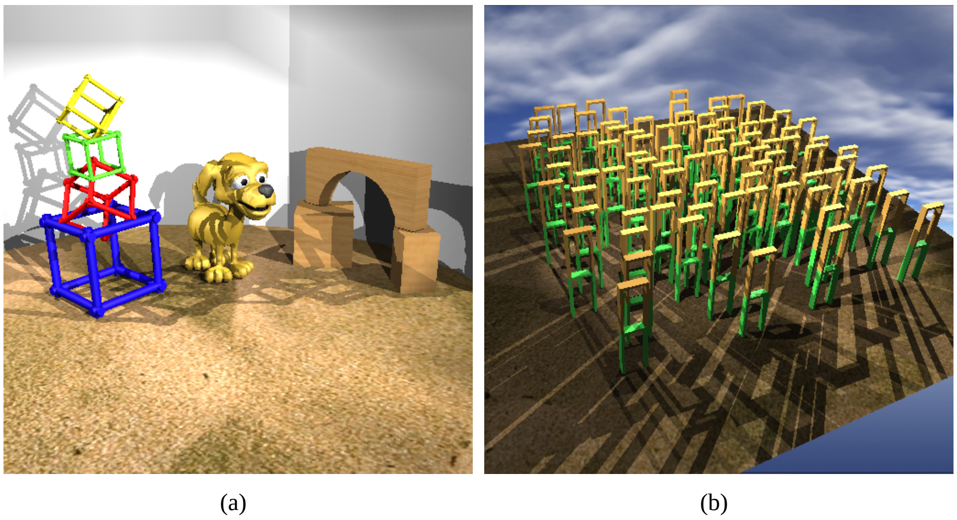


Figure 9.2: Two examples scenes with shadows from three light sources.

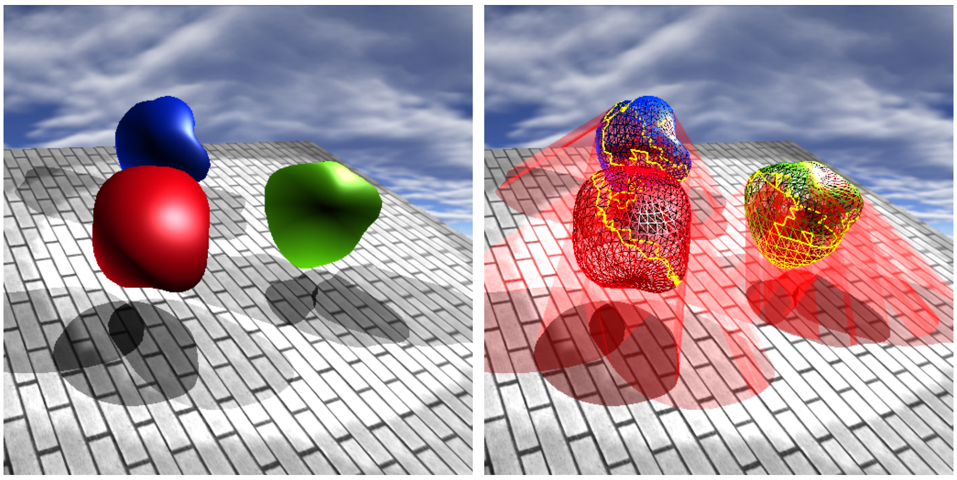


Figure 9.3: Three spheres deformed by a procedural noise shader and illuminated by three light sources.

Part III

Soft Shadow Techniques

Chapter 10

Soft Shadow Maps for Linear Lights

The two previous parts of this thesis showed several ways of efficiently computing shadows caused by point light sources. All methods presented so far are based on either the shadow volume method [Crow77] or the shadow map [Williams78] technique.

Variants to produce soft shadows for linear and area light sources are known both for the shadow volume and for the shadow map algorithm (see, for example [Bergeron86, Brotman84], discussed in Chapter 3). These work by replacing the linear or area light source with a number of point light sources. In many cases, the light source does not subtend a very large solid angle as seen from any object point in the scene. This means that, especially in scenes with mostly diffuse materials, the local illumination caused by different samples of the light source differs only marginally, and thus a small number of light source samples should be sufficient. Nonetheless, the number of samples often has to be quite significant to obtain smooth penumbra regions. This is due to the fact that, with N light source samples, we obtain $N+1$ different levels of shadow: fully lit, fully shadowed (umbra), as well as $N-1$ levels of penumbra. Thus we will need to have a large number of light source samples for scenes with large penumbra regions, or the quantization into $N-1$ penumbra regions will become apparent.

In other words, while a small number of samples would be sufficient for the local shading process, we require a large number of samples to establish the correct visibility in the penumbra regions. This significantly increases the computational cost of soft shadows, and makes them infeasible for many interactive applications.

In this chapter, we introduce a new soft shadow algorithm based on the shadow map technique. This method is designed to produce high-quality penumbra regions for linear and area light sources with a very small number of light source samples. It is not an exact method and will produce artifacts if the light source is so severely undersampled that the visibility information is entirely insufficient (i.e. if there are some portions of the scene that should be in the penumbra, but are not seen by any of the light source samples). On the other hand, it produces believable soft shadows as long as the sampling is good enough to avoid these problems. Figure 10.1 gives a first example of our technique.

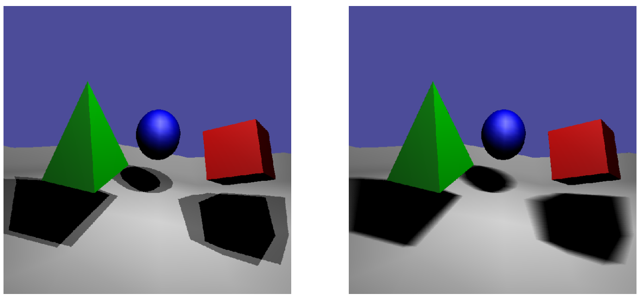


Figure 10.1: Left: approximating a linear light source with two point lights. Right: our method, also using two light source samples.

10.1 Soft Shadow Maps

We start the discussion of our soft shadow algorithm by considering the simple case of a linear light source. One property of such a light is that edges parallel to a linear light source do not have a penumbra region. In other words, there is a sharp transition from umbra to fully lit regions for these edges. Furthermore, linear light sources have the advantage that the visibility considerations of a 3D scene can be reduced to 2D scenes. Consider the intersection of the scene with a plane containing the light source. If we can solve the visibility problem for all such planes, i.e. for the whole bundle of planes having the light source as a common line, then we know the visibility of the light source for all 3D points in the scene.

For a motivation of our soft shadow algorithm, consider the configuration in Figure 10.2, which contains a linear light source at the top, an occluder and a receiver polygon. In order to compute the correct penumbra, we have to determine for each point on the receiver, which percentage of the linear light source is visible from that point. This percentage is plotted as a function of the surface location at the bottom of Figure 10.2.

In this simple configuration, it is clear that we have two penumbra regions, one where the visibility varies from 100% at $\mathbf{p}_1$ to 0% at $\mathbf{p}_2$, and similarly from 100% at $\mathbf{q}_1$ to 0% at $\mathbf{q}_2$. In general, the transition from fully visible to fully occluded is a rational function, which becomes obvious by considering the simple case of a single occluder edge, as depicted in Figure 10.3.

Without loss of generality, the occluder edge is located at the origin (the slope of the occluder is not of importance), the light source is given by the formula $y_1 := mx_1 + t$, and the intersection of the receiver with the 2D plane in consideration is given by $y_2 := nx_2 + s$. From the constraint $x_1/x_2 = y_1/y_2$, which characterizes

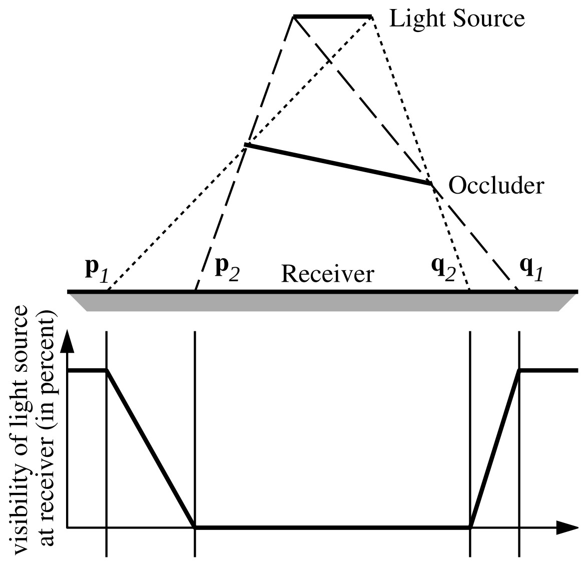


Figure 10.2: Top: a simple scene with a linear light source, an occluder and a receiver polygon. Bottom: the percentage of visible portions of the light source as a function of the location on the receiver.

the point (x_1,y_1) on the light source that is just visible from (x_2,y_2), it follows that

$$x_1 = \frac{x_2 t}{n x_2 - m x_2 + s}. \qquad (10.1)$$

This rational function simplifies to a linear one if the slopes m of the light source and n of the receiver are identical, i.e. if light and receiver are parallel as in Figure 10.2.

If light source and receiver are not parallel, the rational function has a singularity at the point where the receiver polygon intersects the line on which the linear light source resides. However, this is an area where the penumbra region collapses to zero size anyway. On the other hand, the regions for which we expect large penumbra regions are far away from this singularity, and there the rational function from Equation 10.1 behaves almost like a linear function.

Following these thoughts, the process of computing the soft shadows has now been reduced to finding the value of a linear approximation of the visibility function for each point on the receiver, as depicted in Figure 10.2. We would like to do this with a shadow map algorithm, by replacing the linear light source with a small number, say two, point lights. Of course, the shadow map for each individual point light does not have any information about the penumbra regions in the scene.

However, we can texture map the visibility information onto the scene by adding a second channel to the two shadow maps corresponding to the two point lights. Like in the original algorithm, the first channel stores the reference depth

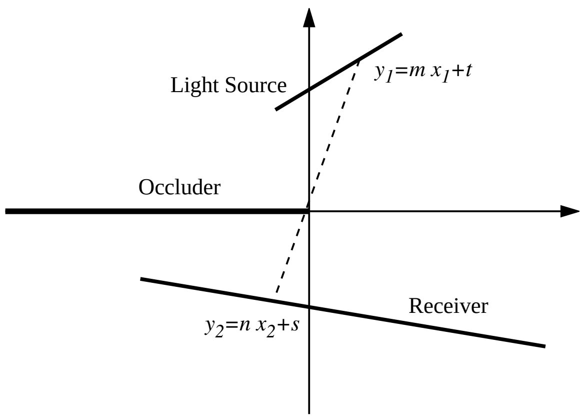


Figure 10.3: A simple scene with a single occluder edge that can be used to characterize the change of visibility across a planar receiver that is not parallel to the linear light source.

values of objects as seen from a particular point light. The second, new channel contains visibility values between 0 and 1 for each of the object points visible from the corresponding point light. Put differently, the shadow map not only contains information about *which* object points are visible from a given point light, but also a percentage value that describes *how much* of the whole linear light source can be seen by that point. For any given object point, the sum of these visibility terms from the two point lights should then result in the value of the function plotted at the bottom of Figure 10.2.

Given two shadow maps S_1 and S_2 including such visibility channels V_1 and V_2, one for each of the two point light sources L_1 and L_2, the shading of a particular point **p** in the scene proceeds according to the following algorithm:

```
shade( $\mathbf{p}$ ) {
  if( depth($\mathbf{p}$)> $S_1[\mathbf{p}]$ )
    l1= 0;
  else
    l1= $V_1[\mathbf{p}]$ * localIllum($\mathbf{p}$,$L_1$);

  if( depth($\mathbf{p}$)> $S_2[\mathbf{p}]$ )
    l2= 0;
  else
    l2= $V_2[\mathbf{p}]$ * localIllum($\mathbf{p}$,$L_2$);

  return l1+l2;
}
```

In this piece of pseudo code $S_i[\mathbf{p}]$ means looking up the reference depth value corresponding to **p** in shadow map *i*. Similarly $V_i[\mathbf{p}]$ means looking up the visibility value for **p**.

10.1.1 Generating the Visibility Map

The remaining question is, how to generate the visibility channels for the two shadow maps. The object points in one of the penumbra regions are of particular interest. In our simple setting, these are the object points seen by one of the two point lights, but not by both.

Now imagine we take the shadow map from the right sample point, triangulate all the depth samples, and warp all the resulting triangles into the view of the left point light, thereby using the depth buffer to resolve visibility conflicts. This is similar to an image based rendering algorithm along the lines of post-rendering 3D warping [Mark97]. The resulting image will consist of two kinds of polygons: those corresponding to the real geometry in the scene, and "phantom polygons", sometimes also called "skins", which result from triangulating across depth continuities. Both types are depicted in Figure 10.4. The skins are depicted as gray lines, while the original surfaces are colored black.

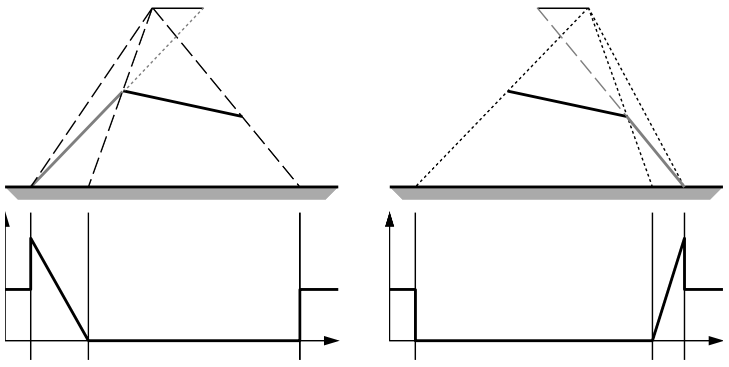


Figure 10.4: Top: skin polygons warped from one depth map into the other. Bottom: visibility contributions for both point lights at each point on the receiver, using the presented method to generate the visibility channel of the shadow maps.

While the original polygons are the desired result in IBR and the skins are an artifact, it is the skins that are of particular interest to us. Wherever they are visible in the destination image (i.e. in the image corresponding to the left point light), a penumbra region is located! Moreover, we know qualitatively what the visibility value should be for points in this region. Since the skins are generated by depth discontinuities in the source shadow map, they always connect an occluder polygon and a receiver polygon. Points in the penumbra region that are closer to the occluder in the reprojected image see less of the linear light than points closer to the receiver polygon.

If we assume a linear transition between fully visible and fully occluded, as argued in the previous section, then we can generate the visibility channel as follows: First, we need to find the depth discontinuities in the shadow map of the right point light, which can be done using standard image processing techniques [Gonzalez92], and can be performed at interactive speed. The resulting skins then need to be reprojected into the shadow map of the left point light, and scan converted using a depth buffer algorithm. At the same time we Gouraud-shade the skin polygons by assigning the value 0 to vertices on the occluder and the value 1 to vertices on the receiver. We repeat the whole procedure to project the discontinuities from the depth buffer of the left point light to the right shadow map.

A final consideration for the generation of the visibility channel is the treatment of completely lit and completely shadowed object points. The latter case is simple. Since points in the umbra are not seen by any of the two light sources, they will fail the shadow map tests for both point light sources, and therefore be rendered black (or with an ambient color only). Completely lit points on the other hand, are seen by both point lights, and therefore we need to make sure that the visibilities for both lights sum up to 1. One way of doing this is to give these points a visibility of 0.5 in both shadow maps. This can easily be implemented by initializing the visibility of each point to 0.5 before starting to warp the skin polygons.

Figure 10.4 shows the contributions in visibility to each point on the receiver from Figure 10.2 for the two point lights, using the just described algorithm to generate the visibility channels. Furthermore, Figure 10.5 shows the result of applying this algorithm to a simple 3D test scene.

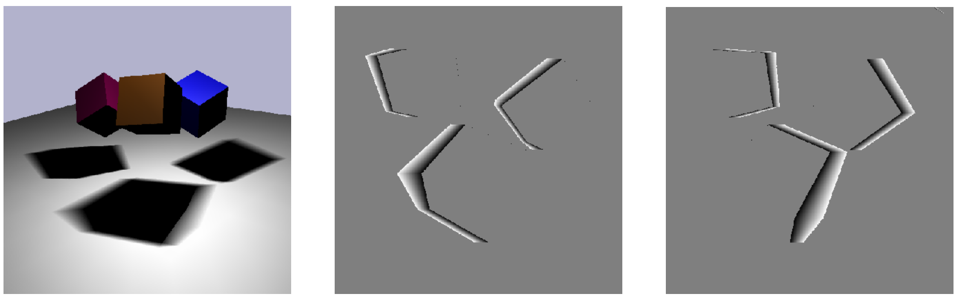


Figure 10.5: Center and right: the visibility channel for the two point lights for the scene on the left.

10.1.2 Linear Light Sources With More Samples

The restriction of this algorithm for generating the visibility map is that object points seeing portions of the linear light source, but none of the two point lights at its ends, will appear to lie in the umbra. Moreover, there are situations where this results in discontinuities, as depicted in Figure 10.6. These artifacts result from a severe undersampling of the light source, with the consequence that important visibility information is available in neither of the two shadow maps.

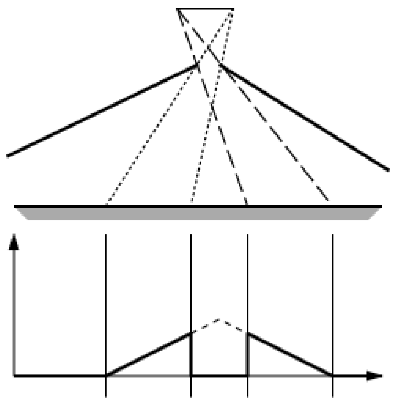


Figure 10.6: An example of failure due to extreme undersampling of the light source which causes some portions of the penumbra to end up in full shadow. These artifacts can only be resolved by increasing the sampling rate on the light source.

The consequence from this observation is to increase the sampling rate by adding in one or more additional point lights on the linear light source. For example, if we add in a third point light in the center of the linear light, we have effectively subdivided the linear light into two smaller linear lights that distribute only half the energy of the original one. If we treat these two linear light segments independently with the algorithm described in the previous section, we get the situation depicted in Figure 10.7 for the same geometric setup as in Figures 10.2 and 10.4.

The top row of the figure corresponds to the rendering of the left half, while the bottom row corresponds to the right half of the linear light. Note that the light source on the right side of the top row, and the one on the left side of the bottom row correspond to the same point light, namely the one inserted at the center of the linear light. Therefore it is possible to combine these two point lights into a single one with twice the brightness, by summing together the visibility channels (the depth channels are identical anyway!).

With this general approach we can add in as many additional sample points on the linear light source as are required to avoid the problems of points in the penumbra that are not seen by any light source sample. To generate the visibility channel for one of the sample points, we need to consider only the depth discontinuities (skins) of those samples directly adjacent to this point. For example, in Figure 10.7, the discontinuities from the rightmost sample do not play a role for the visibility map of the leftmost sample and vice versa.

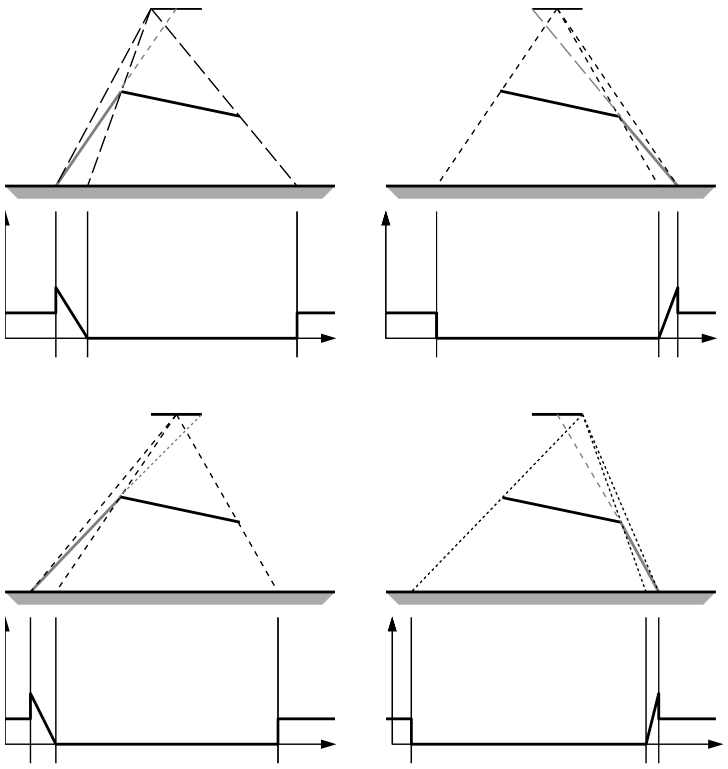


Figure 10.7: By inserting an additional point light in the center, we have effectively reduced the problem to two linear lights of half the length and intensity. Top row: left half of linear light. Bottom row: right half.

10.1.3 Extension to Area Light Sources

So far, we have exclusively considered linear light sources. Area light sources provide even more complex and interesting penumbra regions. A simple extension of the algorithms presented in the preceding two sections to area light sources works as follows.

We start with a light source consisting of a single triangle. Like in Section 10.1.1 we can simply generate the depth maps for each of the vertices of the triangles, which is where we place point lights to substitute for the triangular light source. By finding the discontinuities in these depth maps, generating the corresponding skins, and warping them to both of the other maps, we can again mark the positions of the penumbra regions in the respective map.

At this point, some more sophisticated analysis of the geometric configuration would be necessary to yield a better approximation for the visibility transition between fully visible and fully occluded. For area light sources, this transition is, in general, a quadratic rational function, so that the linear approximation used for linear light sources may no longer be adequate. In our work and in all the examples we present here we ignore this fact and use the linear approximation anyway. The results are surprisingly convincing, as you can see in Section 10.2.

On the other hand, if we had a better approximation of the true function, which would have to be developed as part of future work, we could integrate it by putting it into a texture, and applying it to the skin polygons while those are warped to the respective destination shadow map. This would then replace the Gouraud-shading we currently use. From this point on, the adaptation of the algorithm to more complex area light sources comprised of a mesh of triangles, is simple, and proceeds as described in Section 10.1.2. As mentioned there, we can again merge the visibility channels of those maps that belong to the same vertex (i.e. point light) in the triangle mesh. As a result, the number of shadow map tests is identical to the number of vertices in the triangle mesh.

A special variant of our method is presented by Ying et al. [Ying02]. Here the authors use the information stored in the visibility channel to approximate the area of the light source as seen from a given sample point. This is illustrated in Figure 10.8: Given a polygonal, convex light source, we run our algorithm for each edge of the polygon. This results in a total of five shadow maps (one for each sample point) and ten visibility maps (two for each edge) that have to be generated.

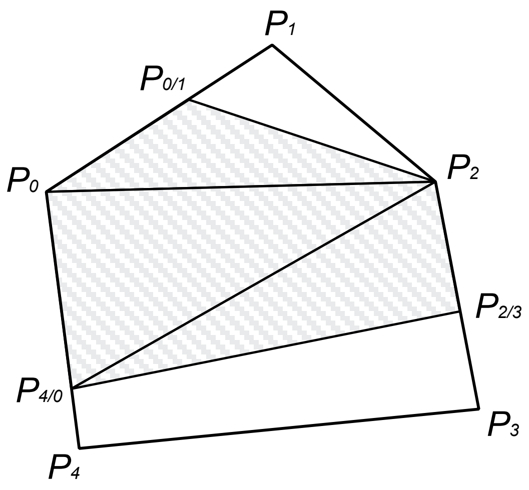


Figure 10.8: Computing the visibility of a polygonal, convex light source.

Using the shadow maps, we observe that the sample points P_0 and P_2 are blocked, whereas the other three sample points can be seen. For each edge that connects a visible and blocked sample point ($[P_0P_1], [P_2P_3], [P_4P_0]$), the visibility channel of the visible sample point (P_1, P_3, P_4) tells us at which fraction the given edge is divided ($P_{0/1}, P_{2/3}, P_{4/0}$). Given this information, it is now an easy task to compute the visible area of the light source.

10.1.4 Soft Shadow Map Implementation

Our soft shadow map technique is based on the color-coded shadow map implementation presented in Section 3.2.4. Color-coded shadow maps store the depth value in one (or more) color channels of a regular texture map. This can be achieved by using automatic texture coordinate generation and a 1D ramp texture which maps the r component (which corresponds to the z value) to the color channels. In our implementation we restrict ourselves to 8-bit depth precision and use the texture's alpha channel to store the depth value.

If we have an additional visibility channel as described in Section 10.1, we store it in the luminance channel of the shadow map, so that each map is then a luminance-alpha texture. We then render the contributions from the individual light source samples in separate rendering passes, and add up the results. The pass for each individual light is not much different from the standard shadow map algorithm. The only change to the algorithm for point lights is that we have to set up OpenGL to multiply the luminance channel of the visibility channel with the result from the OpenGL lighting (fragment color). This does not require an additional rendering pass.

What remains to be done is the generation of the visibility channel. It closely follows the description in Section 10.1.1. In order to find the discontinuities in the depth maps, we use convolution with a 3×3 Laplacian edge detection filter followed by a thresholding operation. Both operations are also supported by the OpenGL imaging subset. However, the generation of the geometry for the skin polygons from the edge enhanced shadow map can only be performed in software. This requires reading back the framebuffer into main memory, which may cause a performance bottleneck on low-end systems.

10.2 Results

We have implemented the approaches described in this chapter for two different hardware platforms. Firstly, we use SGI workstations (SGI Octane, O2 and Visual Workstation), which all have support for the OpenGL imaging extensions, but not for features like multitexturing that would help us reducing the number of rendering passes.

Secondly, we have used an NVIDIA GeForce 256 with DDR RAM. This graphics board supports multitexturing with a flexible way of combining the resulting colors for each fragment. This allows us to implement the core shadow map algorithm in a single rendering pass. On the other hand, this chip set, like the other consumer graphics boards, does not yet support the OpenGL imaging extensions, so that the required features had to be implemented in software. This is slower than the hardware implementation on the workstations for two reasons. Firstly, the operations are not parallelized as in the hardware versions, and secondly, we need to read back the framebuffer contents over the AGP bus to perform the operations. The resulting bandwidth is the bottleneck of the implementation.

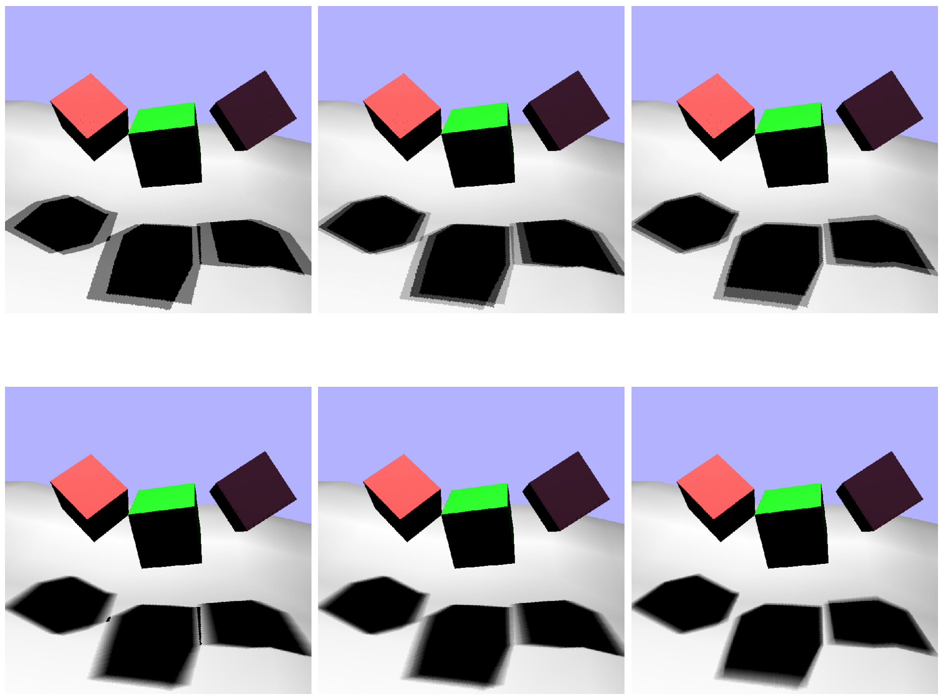


Figure 10.9: Comparison of different soft shadow techniques. Top row: simple approximation of the light source by several point lights. Bottom row: the method proposed in this chapter. Left column: a linear light source approximated with two samples. Note the artifact introduced where the soft shadows overlap. This is due to undersampling (see Section 10.1.2). Center: The artifacts disappear as a third light source sample is introduced. Right: Result from a triangular area light source.

A comparison of the different variants of our soft shadowing algorithm is depicted in Figure 10.9. The top row shows the images that would be generated by simply approximating a linear or area light source with a number of point lights. The bottom row shows results from our algorithm with the same number of light source samples. It can be seen that the quality of the penumbra regions is much higher in all cases. The left column shows the result from approximating a linear light source with two samples. It has been chosen such that overlapping penumbra regions exhibit the undersampling artifacts described in Section 10.1.2. These artifacts disappear as a third light source sample is inserted, as shown in the center column. Finally, the right column depicts images taken with a triangular area light source. Although we only use a linear transition between umbra and fully lit regions, as described in Section 10.1.3, the penumbrae look quite convincing.

Figure 10.10 compares a high-quality solution for the visibility of a scene with one linear light source, one blocker and one receiver with our method. Figure 10.10a shows the solution of a ray-tracer using 200 light source samples to determine the visibility in every point on the receiver. Figure 10.10b depicts the result of a software implementation of our method. In contrast to the OpenGL implementation, the software implementation allows for having the same per-pixel shading as in the ray-traced image. Figure 10.10c shows a ray-traced solution with 10 uniformly spaced samples for comparison. With a shadow map resolution of 500×500, our method including map generation and rendering of 2×1700 skin polygons takes about as long as ray-tracing with 6 samples.

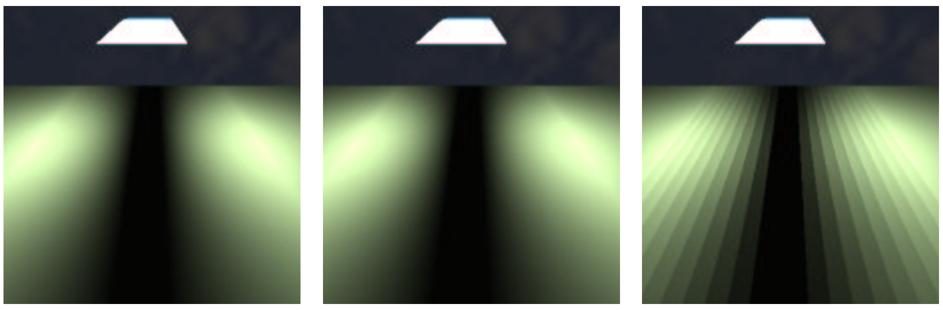

(a) ray traced, 200 samples (b) our method, 2 samples (c) ray traced, 10 samples

Figure 10.10: A comparison of ray-traced images and our method.

Figure 10.11 shows some more complex scenes. Once the shadow maps are computed, the rendering times using our soft shadow algorithm are identical to those for rendering hard shadows with the same number of point lights. This is true for all scenes. Therefore, our algorithm can be used for interactive walkthroughs with no additional cost.

Building the shadow maps in a dynamic environment is obviously more expensive for our algorithm, since the visibility channels need to be generated as well. This requires edge detection within the depth maps, as well as rendering a potentially large number of skin polygons. The cost of generating the shadow maps therefore depends on the scene geometry. It varies from $< 1/20$ seconds for the simple scene in Figure 10.9 to about 2 seconds for the area light source in the jack-in-a-box scene in Figure 10.11. These numbers include the time for rendering the scene to generate the depth maps.

10.3 Discussion

In this chapter we presented a new soft shadow algorithm based on the shadow map method. It is designed to produce high-quality penumbra regions for linear and area light sources with a small number of light source samples. We demonstrated that the method works efficiently and produces high-quality penumbrae for non-trivial scenes. Furthermore, we showed that the method can efficiently utilize graphics hardware for interactive display.

It remains an open research problem to determine the best place to insert samples into a linear or area light source. Work by Ouellette and Fiume [Ouellette99a] seems to be a promising starting point for determining those locations where a new sample point would improve the overall quality of the penumbra regions the most. Furthermore, the shading of the penumbra for the case of area light sources deserves more attention. A linear visibility transition, as used in our method, is not always a good assumption for this case.

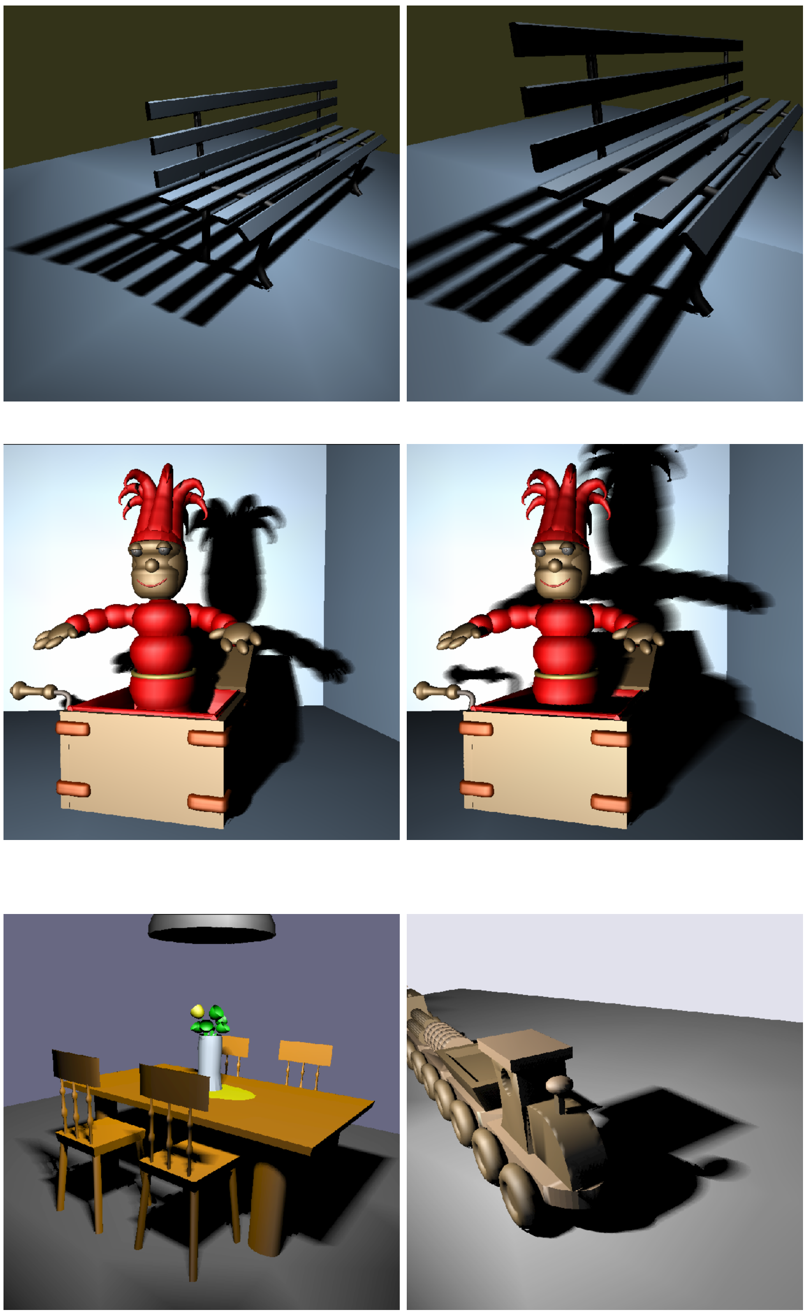


Figure 10.11: Some more examples.

Chapter 11

Single Sample Soft Shadows using Depth Maps

As we have seen in the last chapter, the efficient computation of soft shadows is a very complicated task. The soft shadow maps method described previously reduces the workload by using only a small number of sample points, e.g. the two endpoints of a linear light source. To approximate the penumbra caused by an area light source, even more sample points have to be considered in order to achieve realistic shadowing effects.

However, for real-time applications, such as computer games, we often do not necessarily need a physically correct solution rather than a visually pleasing result, which should give the impression of a *correct* soft shadow computation.

In this chapter we will present a way of computing these kind of *fake* soft shadows using only sampled images taken from the view of a point light source. This soft shadow algorithm can be seen as an extension of the classical shadow map algorithm for calculating hard shadows. Instead of computing only a binary value (shadowed or lit) for each pixel seen by the camera, our algorithm processes the neighborhood of the corresponding depth map entry to gather information about what the shadow might look like in the case of an area light source.

Even though the input data contains no information about the characteristics of an area light, the resulting shadows are yet of very good quality and give the impression of a physically plausible computation. Using only a minimal amount of input data and a very compact algorithm, we can achieve extremely high computation speed. This way we can also utilize graphics hardware and specialized processor instruction sets.

11.1 Single Sample Soft Shadows

Parker et al. [Parker98] showed how soft penumbra regions can be generated by defining an extended hull for each possible occluder object. By treating the *inner* object as opaque and having the opacity of the *outer* object fall off towards the outer boundary one can dim the contribution of a point light source according to the relative distances of light, occluder, and receiver. Figure 11.1 illustrates this scheme.

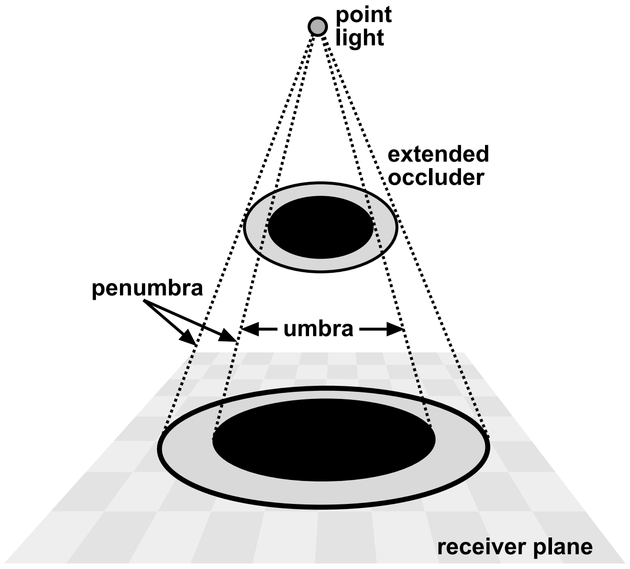


Figure 11.1: Computing penumbrae for a point light source.

In order to avoid light leaks occurring for adjacent objects the size of the inner object needs to be at least as large as the original occluder geometry. Since this causes relatively large umbra regions, which would not occur in a physically correct shadow computation, the approximation still produces reasonably looking shadows as long as the occluder objects aren't too small compared to the simulated light source area. Parker et al. implemented this scheme using standard ray tracing. In this case, it is a comparatively easy task to compute the extended hulls for primitives like spheres and triangles, and ray intersection directly calculates the distances to the outer and inner boundaries, which are used to compute a corresponding attenuation factor.

Although it was shown that the algorithm only introduces about 20% of computation overhead (compared to normal ray tracing), it is still far from being suitable for interactive rendering. Especially when it comes to more complex scenes, too much computation is spent on extending the geometric primitives and computing attenuation factors that later will be discarded.

In the following sections we will show that this method can be adopted to work on sampled data (depth maps) in a much more efficient manner, while still achieving good shadow quality.

11.2 A Sampling Based Approach

Just like the traditional shadow map algorithm presented in [Williams78], we start with the computation of two depth images, one taken from the view of the point light source and one taken from the camera. To compute hard shadows we simply have to compare the transformed z value of each frontmost pixel (as seen by the camera) to the corresponding entry in the light source depth map, according to the algorithm shown in Figure 11.2.

```
foreach($x$,$y$) {
  $P = (x, y,$ depth_camera$[x, y])$
  $P'$ =warp_to_light($P$)
  if(depth_light$[P'_x, P'_y] < P'_z$)
    pixel is blocked
  else
    pixel is lit
}
```

Figure 11.2: Shadow map algorithm.

To modify this method to add an *outside* penumbra region, we have to extend the `else` branch of the shadow test to determine if the pixel is really lit or lies in a penumbra region. According to the ray tracing scheme explained in the previous section, we have to trace back the ray from the surface point towards the light source and see if any outer object is intersected. If we consider the light source depth map as a collection of *virtual layers*, where each layer is a binary mask describing which pixels between the light and the layer got blocked by an occluder inbetween (hard shadow test result), we can simulate the intensity fall-off caused by an area light source by choosing the nearest layer to P'_z that is still in front, and compute the distance between (P'_x, P'_y) and the nearest blocked pixel in that specific layer. This is in a sense similar to Parker's method since finding the minimum distance corresponds to intersecting the outer hull and computing the distance to the inner boundary. The main difference is of course that we use a sampled representation containing all possible occluders rather than the exact geometry of only one occluder.

Figure 11.3 illustrates the search scheme using a very simple setup consisting of the umbra generated by an ellipsoid as an occluder and a ground plane as the receiver polygon. For a given point P which does not lie inside the umbra, we first warp P to the view of the light source (P'). Since the original point P was somewhere near the umbra, we find the transformed point P' in the neighborhood of the blocker image which causes the umbra. To calculate an attenuation factor for P, we start searching the neighborhood of P' till we either find a blocked pixel or a certain maximal search radius r_{max} is exceeded. The attenuation factor f is now

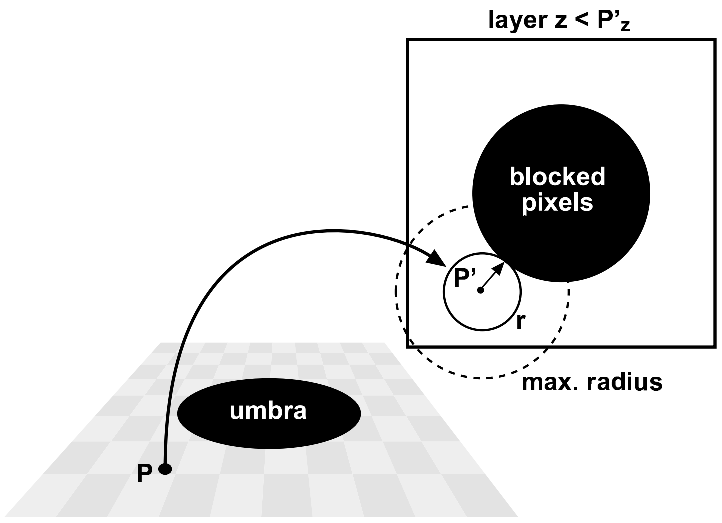


Figure 11.3: Projecting and searching for the nearest blocker pixel.

simply the minimal distance r (or $r = r_{max}$ if no blocking pixel is found) divided by the maximal radius r_{max}. So $f = \frac{r}{r_{max}}$ rises up from 0 (no illumination) to 1 (full illumination) as the distance to the blocker increases.

In other words, we can now generate smooth shadow penumbra regions of a given maximal extent r_{max}. To simulate the behavior of a *real* area light source, we now have to define which properties affect the size of the penumbra and how these can be realized with our search scheme. As widely known, the following two distances mainly define the extend of the penumbra[1]:

- the distance between occluder and receiver, and
- the distance between receiver and light source.

For our search scheme the distance between receiver and light source can be integrated by varying r_{max} according to the distance between a given surface point P and the light source position. Assuming a fixed occluder, a receiver far away from the light source will get a larger penumbra whereas a receiver near to the light source will have a smaller r_{max} assigned.

Taking into account the distance between occluder and receiver is a little bit tricky: Since finding an appropriate occluder is the stop criterion for our search routine, we do not know in advance what this distance will be. What we do know is that the occluder has to be inside the region determined by the maximal extend, which is computed using the distance between receiver and light source.

In other words, the final r_{max} may be less than the initial search radius. For our search routine this means that we first search the maximal extend since an occluder

[1]Apart from other properties like orientation of receiver and light source etc.

pixel is found and then re-scale the initial search radius by a factor computed using the distance between the surface point P and the found occluder pixel and use this r_{max} as the denominator for computing f (attenuation factor).

Assuming that the position of the point light in light source space is located at $(0,0,0)$ and that the light direction is along the z axis, we set the initial search radius

$$r_{max} = r_{scale} * |P'_z| + r_{bias},$$

where r_{scale} and r_{bias} are user defined constants describing the area light effect[2]. Since shadow maps are usually generated for the very limited cut-off angle of spotlights, the difference of using P'_z instead of computing an euclidean distance is negligible. We can now rewrite the hard shadow algorithm to produce soft shadows by simply adding this search function (see Figure 11.4).

```
foreach($x$,$y$) {
  $P = (x, y,$depth_camera$[x,y])$
  $P'$ =warp_to_light($P$)
  if(depth_light$[P'_x, P'_y] < P'_z$)
    pixel is blocked
  else
    $f =$ search($P'$)
    modulate pixel by $f$
}
search($P'$) {
  $r = 1$
  $r_{max} = r_{scale} * |P'_z| + r_{bias}$
  while($r < r_{max}$) {
    if $\exists(s,t) : \|(P'_x, P'_y) - (s,t)\| = r \wedge$
        depth_light$[s,t] < P'_z$ {
      $r_{max} *= r_{shrink} * (P'_z -$ depth_light$[s,t])$
      return $clamp_{0,1}(r/r_{max})$
    }
    else
      increase $r$
  }
  return 1.0
}
```

Figure 11.4: Soft shadow algorithm.

In the first loop we iterate over all frontmost pixels as seen by the camera performing the hard shadow test. For each lit pixel we start a search routine where

[2] r_{bias} can be used to force a certain penumbra width even for receivers very near to the light source.

we search in the light source depth map in order to find a suitable blocker pixel at a minimal distance to the transformed surface point. If a blocker pixel is found we then re-scale the initial r_{max} by a factor computed using the distance between the surface point and the occluder pixel. An user-defined scaling factor r_{shrink} is used to give additional control on the effect of this distance.

As can be seen in the pseudo code the described *virtual layers* are implicitly selected by processing only those pixels in the depth map where a blocker lies in front of the potential receiver ($\texttt{depth\_light}[s,t] < P'_z$).

Up to now we have restricted ourselves to a very simple setup where the receiver was parallel to the light source image plane. This has the effect that P'_z remains constant during the soft shadow search, or in other words, the search takes place in a constant virtual layer. This is no longer the case if we consider an arbitrary receiver as depicted in Figure 11.5.

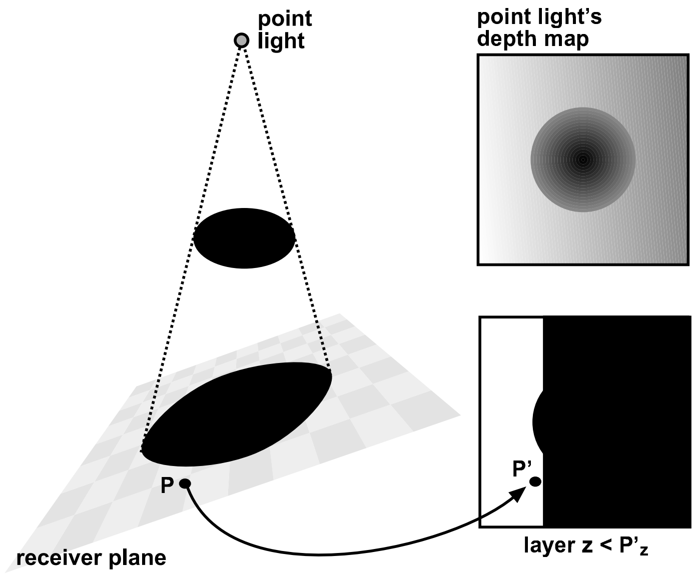


Figure 11.5: Wrong self shadowing due to constant z**.**

If we performed a search on the constant layer $z < P'_z$ we would immediately end up in the receiver's own shadow since the receiver plane may cross several of the virtual layers. This can be seen in the virtual layer image in Figure 11.5 where about two thirds of the layer contain blocked pixels belonging to the receiver polygon.

To solve this problem, we either have to divide the scene into disjunct occluders and receivers[3], which would make the algorithm only suitable for very special situations, or we need to supply more information to the search routine. To define an additional search criterion, which gives the answer to the question "does this

[3]Which is e.g. suitable for games where a character moves in a static environment.

blocker pixel belong to me?", we follow Hourcade's [Hourcade85] approach and assign object IDs. These IDs are identification numbers grouping logical, spatially related, objects of the scene.

It must be pointed out that all triangles belonging to a certain object in the scene must be assigned the same object ID, otherwise self shadowing artifacts would occur if the search exceeded the projected area of the triangle belonging to P. Of course there are situations where also the ID approach fails, e.g. if distinct objects are nearly adjacent, but for most real-time applications there should exist a reasonable grouping of objects.

11.2.1 Handling of Hard Shadow Regions

Up to now we have concentrated on the computation of the *outer* part of the hard shadow region and simply assumed that the hard shadow region is not lit at all. In the case of an area light source, which we would like to simulate, this is of course an indefensible assumption. What we would like to obtain is of course a penumbra region which also smoothes this *inner* region. This can be easily achieved if we apply the same search technique for pixels that are initially blocked by an occluder. Instead of searching for the nearest blocker pixel within a given search radius we now have to search for the nearest pixel that is lit by the light source.

To combine this with the *outer* penumbra result we assume that *outer* and *inner* regions meet at an attenuation value of 0.5 (or some user defined constant). The final algorithm (including the object ID test) that produces penumbra regions can then be implemented according to pseudo code shown in Figure 11.6.

11.3 Implementation

11.3.1 Generating the Input Data

Since our algorithm relies on sampled input data, graphics hardware can be used to generate the input data needed for the shadow computation. In a first step we render the scene as seen by the light source and encode object IDs as color values. For very complex scenes we either use all available color channels (RGBA) or restrict ourselves to one channel (alpha) and assign object IDs modulo 2^n (n bits precision in the alpha channel). This gives us the depth map (z-buffer) and the object IDs of the frontmost pixels according to the light source view, which we transfer back to the host memory. We then repeat the same procedure for the camera view. If only the alpha channel is used for encoding the object IDs, we can combine this rendering pass with the rendering of the final scene (without shadows).

In cases where 8 bits are enough we could also use a special depth/stencil format available on newer NVIDIA GeForce cards. With this mode we simply encode IDs as stencil values and obtain a packed ID/depth map (8 bits stencil, 24 bit depth) using only one framebuffer read. Another benefit of this format is that memory accesses to id/depth pairs are more cache friendly.

```
foreach($x,y$)
{
    $P = (x,y,\texttt{depth\_camera}[x,y])$
    $P' =$warp_to_light($P$)
    $P'_{ID} =\texttt{id\_camera}[x,y]$
    $inner = \texttt{depth\_light}[P'_x,P'_y] < P'_z$
    $f =$ search($P'$, $inner$)
    modulate pixel by $f$
}

search($P'$, $inner$)
{
    $r = 0$
    $r_{max} = r_{scale} * |P'_z| + r_{bias}$
    while($r < r_{max}$)
    {
        if $inner$
        {
            if $\exists(s,t) : \|(P'_x,P'_y) - (s,t)\| = r \quad \wedge$
                  $\texttt{depth\_light}[s,t] >= P'_z \quad \wedge$
                  $\texttt{id\_light}[s,t] == P'_{ID}$
            {
                $r_{max} *= r_{shrink} * (\texttt{depth\_light}[s,t] - P'_z)$
                return $clamp_{0,0.5}(0.5 * (1 - (r/r_{max})))$
            }
        }
        else
        {
            if $\exists(s,t) : \|(P'_x,P'_y) - (s,t)\| = r \quad \wedge$
                  $\texttt{depth\_light}[s,t] < P'_z \quad \wedge$
                  $\texttt{id\_light}[s,t] \neq P'_{ID}$
            {
                $r_{max} *= r_{shrink} * (P'_z - \texttt{depth\_light}[s,t])$
                return $clamp_{0.5,1.0}(0.5 * (1 + (r/r_{max})))$
            }
        }
        increase $r$
    }
    return $inner$ ? 0.0 : 1.0
}
```

Figure 11.6: Final algorithm (including inner and outer penumbra).

11.3.2 Shadow Computation

The actual shadow computation takes place at the host CPU. According to the pseudo code in Section 11.2.1, we iterate over all pixels seen by the camera and warp them to the light source coordinate system. Next we start searching for either the nearest blocker pixel (*outer* penumbra region) or the nearest pixel that is lit (*inner* penumbra region).

Since memory accesses (and the resulting cache misses) are the main bottleneck, we do not search circularly around the warped pixel but search linearly using an axis aligned bounding box. Doing so we are actually computing more than needed but this way we can utilize SIMD (single instruction, multiple data) features of the CPU, e.g. MMX, 3DNow, or SSE, which allows us to compute several r in parallel. If an appropriate blocking pixel is found (object ID test, minimal distance), we store the resulting attenuation factor for the given camera space pixel. If the search fails, a value of 1.0 or 0.0 is assigned (full illumination, hard shadow).

At the end, the contribution of the point light source is modulated by the attenuation map using alpha blending.

11.3.3 Improvements

Subpixel Accuracy

When warping pixels from camera to light there are two ways to initialize the search routine. One would be to simply round (P'_x, P'_y) to the nearest integer position and compute distances using only integer operations. While this should give the maximum performance, the quality of the computed penumbrae would suffer from quantization artifacts. Consider the case where pixels representing a large area in camera screen space are warped to the same pixel in the light source depth map. Since all pixels will find the same blocker pixel at the same distance, a constant attenuation factor will be computed for the whole area.

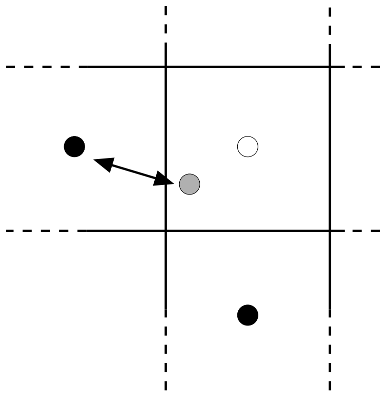


Figure 11.7: Computing distances at subpixel accuracy.

This can be avoided by not rounding to the nearest integer but performing the distance calculation at floating point precision. As depicted in Figure 11.7, we compute the distance of the warped pixel (grey) to the next blocker pixels, which lie at integer position. Quantization artifacts can be further reduced if we also slightly jitter the integer position of the blocker pixels. In practice we observed that the latter is only needed for very low resolution depth maps.

Adaptive Sampling

Up to now we only briefly discussed the cost of searching the depth map. Consider a scene where only 5% of the frontmost pixels are in hard shadow. To compute accurate penumbra regions we would need to perform neighborhood searches for 95% of the pixels in the worst case[4]. So for all completely lit pixels we have searched the largest region (r_{max}) without finding any blocker pixel. Even with a highly optimized search routine and depth maps of moderate size it would be very difficult to reach interactive frame rates.

Instead we propose an interpolation scheme that efficiently reduces the number of exhaustive searches needed for accurate shadowing. The interpolation starts by iterating over the columns of the camera depth map. In each iteration step, we take groups of 5 pixels and do the hard shadow test for all of them. Additionally, we also store the corresponding object IDs of the blockers, or, in the case of lit pixels, the object ID of the receiver pixel. Next, we perform a soft shadow search for the two border pixels in this group. As a criterion for the inner pixels we check if

- the object IDs are equal and
- the hard shadow test results are equal.

If this is true, we assume that there will be no dramatic shadow change within the pixel group and simply linearly interpolate the attenuation factors of the border pixels across the middle pixels. If the group test fails we refine by doing the soft shadow search for the middle pixel and subdivide the group into two three pixel groups for which we repeat the group test, interpolation and subdivision.

Figure 11.8 shows an example of such an interpolation step. Let us assume that the object ID of pixel 3 differs from the rest. In the first phase we perform hard shadow tests for all pixels and soft shadow searches for the two border ones. Since the interpolation criterion fails (IDs not equal), the pixel group is refined by a soft shadow search for pixel 2 and subdivided into two groups. Pixel group $(0,1,2)$ fulfills the criterion and an interpolated attenuation factor is assigned to pixel 1, whereas for pixel group $(2,3,4)$ we need to compute the attenuation by search. As we will later also need object IDs for interpolated pixels, we simply use the object ID of one interpolant in that case. We repeat this for all pixel groups in this and every 4th column, leaving a gap of three pixels in the horizontal direction.

[4]Worst case occurs when all pixels are in the view of the light source.

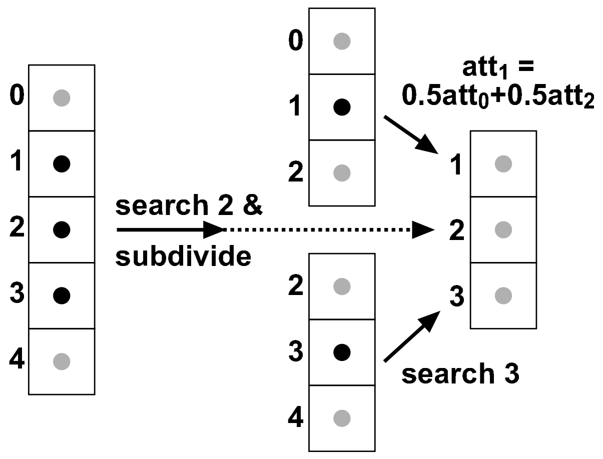


Figure 11.8: Subdivision and interpolation.

Having linearly interpolated over the columns we now process all rows in the same manner and fill up the horizontal gaps. This bi-linear interpolation mechanism is capable of reducing the number of expensive searches. In the best case, the searching is only done for one pixel in a 16 pixel block. Since this case normally occurs very often (e.g. in fully illuminated areas), we can achieve a great speed-up using the interpolation. On the other hand, quality loss is negligible or non-existent because of the very conservative refinement.

The size of the pixel group used for interpolation should depend on the image size. In our experiments we observed that blocks of 4×4 pixels are a good speed/quality trade-off when rendering images of moderate size, e.g. 512×512 or 800×600 pixels, whereas larger block sizes may introduce artifacts due to the non-perspectively correct interpolation.

11.4 Results

We have implemented our soft shadow technique on an Intel Pentium 4 1.7GHz computer equipped with an NVIDIA GeForce3 graphics card. Since the generation of depth and ID maps is done using graphics hardware, we get an additional overhead due to the two frame buffer reads needed to transfer the sampled images back to host memory.

Figure 11.10 shows the results of our soft shadow algorithm for a very simple scene consisting of one torus (occluder) and a receiver plane. We rendered the same scene three times varying only the position and orientation of the occluder.

All images in Figure 11.10 were rendered using an image resolution of 512×512 pixels and a light depth/ID map resolution of 256×256 pixels. By default, we always use the full image resolution when computing the depth and ID map for the camera view. Frame rates for this scene are about $10-15fps$. Computing only the hard shadows (shadow test done on the host CPU) the scene can be rendered at about $30 fps$.

In the left image r_{max} was set to 20 (20 pixel search radius) for the inner and outer penumbra. The receiver plane does not reach this maximum (due to the distance between receiver and light source). The average search radius for pixels on the receiver plane is about 16 pixels. The effect of increasing or decreasing r_{max} for this scene is plotted in Figure 11.9. It must be pointed out that the distance between occluder and receiver does not affect the initial search radius. Therefore the cost of computing soft shadows for the three images in Figure 11.10 is nearly constant.

In the left image artifacts can be seen (ring), where the inner and outer penumbra meet. This is because the attenuation factors for inner and outer regions are computed in a slightly different way (see Section 11.2.1). Theoretically this transition should be smooth.

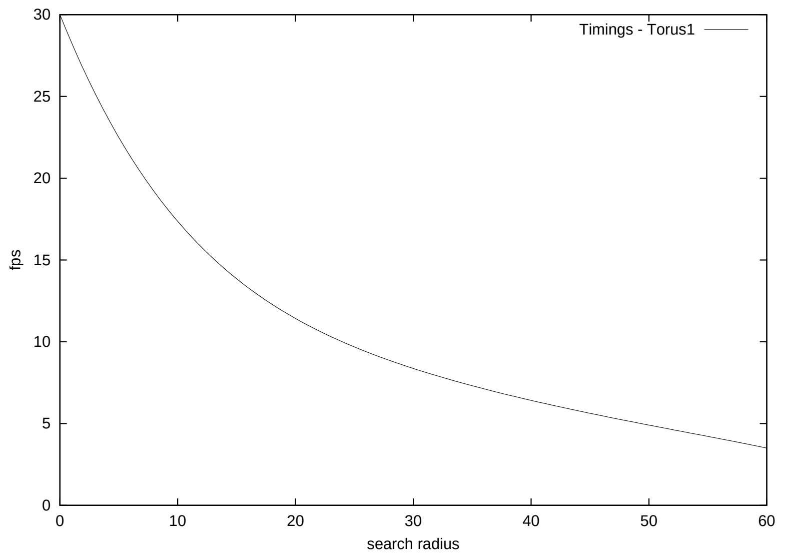


Figure 11.9: Frame rates for the torus test scene (Figure 11.11)

Figure 11.11 (left) shows a more crowded example scene with objects placed at various heights. It can be seen that objects very near to the floor plane cast very sharp shadows, whereas the shadows from the three tori are much smoother. The other two images in Figure 11.11 show the scene with hard shadows and hard shadows with outer penumbra. Since our soft shadow algorithm is based on the shadow map technique, we are independent of the scene geometry, which means we can generate soft shadows for arbitrary geometry. There is no distinction between receiver and occluder objects (apart from the missing self shadowing due to the ID test).

Figure 11.12 shows two more complex scenes where we used our soft shadow algorithm for penumbra generation. In order to assign reasonable object IDs we simply group polygons using the tree structure obtained when parsing the scene file. This way all polygons sharing the same transformation and material node are assigned the same object ID. Both images were taken using a low-resolution light depth/ID map of 256×256 pixels and an image resolution of 512×512 pixel. In the right image we choose a very large cutoff angle for the spotlight which would normally generate very coarse hard shadows. Here the subpixel accuracy explained in Section 11.3.3 efficiently smoothes the shadows. Both images can be rendered at interactive frame rates ($\approx 15 fps$).

Note that all the timings strongly vary with the size of the penumbra, so changing the light position or altering r_{max} may speed up or slow down the computation, depending on the number of searches that have to be performed. When examining the shape of the penumbrae, one can observe that they do not perfectly correspond to the occluder shape. This is due to the circular nature of the search routine, which rounds off corners when searching for the minimal distance.

11.5 Discussion

In this chapter we have shown how good-looking, soft penumbra regions can be generated using only information obtained from a single light source sample. Although the method is a very crude approximation it gives a dramatic change in image quality, while still being computationally efficient. We showed how the time consuming depth map search can be avoided for many regions by interpolating attenuation factors across blocks of pixels. Since the algorithm works on sampled representations of the scene, computation time depends mostly on the shadow sizes and image resolutions and not on geometric complexity, which makes the method suitable for general situations.

In its current state the algorithm still relies on a number of user parameters (r_{max}, r_{shrink}, etc.) which where introduced *ad-hoc*. As future work we would like to hide these parameters and compute them based on one intuitive parameter (e.g. the radius of a spherical light source, defined in the scene's coordinate system). This way it would also be possible to compare our method to more accurate algorithms.

With real time frame rates as a future goal, another focus will be on more sophisticated search algorithms that work on hierarchical and/or tiled depth maps as well as investigating methods of precomputed or cached distance information. Kirsch and Döllner [Kirsch03] introduced so called shadow width maps, which store the distances of blocked pixels to the shadow boundaries by placing a virtual plane behind all occluder objects. During rendering they can then simply lookup a general inner width and modulate it according to the actual distances between light source, occluder, and receiver. However, their method only works for the inner soft shadow region and does not support overlapping shadow regions.

Further speed improvements could also be achieved by using graphics hardware, e.g. interleaved framebuffer reads, as well as on the host CPU by using special processor instructions sets.

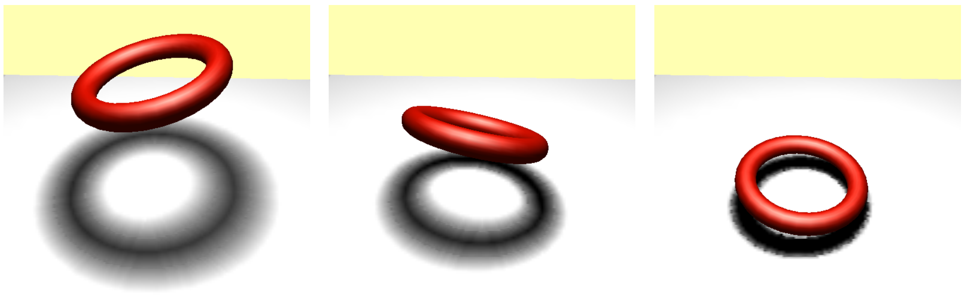


Figure 11.10: A simple test scene showing the effect of varying distance between receiver and occluder.

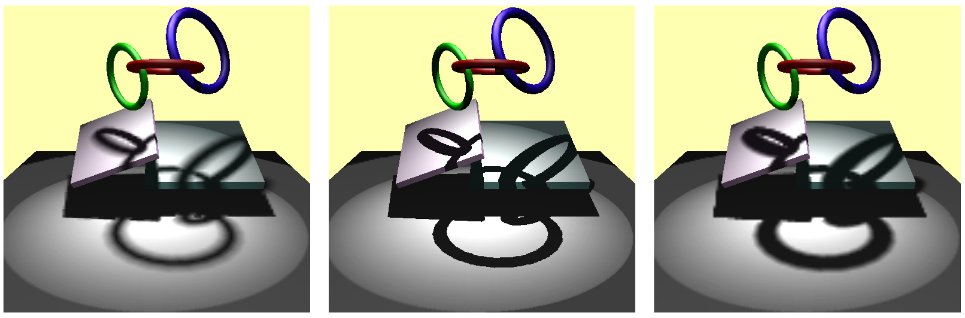
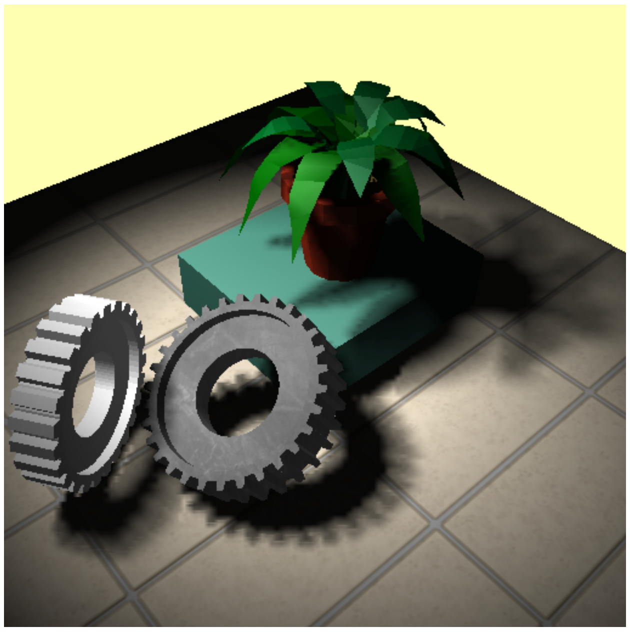


Figure 11.11: A more *crowded* scene. Left: soft shadows, middle: hard shadows, right: hard shadows with outer penumbra.

Another research direction will be the quality of shadows. Up to now we simply used a linear intensity fall-off, which of course is not correct. Assuming a diffuse spherical light and an occluder with a straight edge (similar to Parker's original algorithm), a better approximation would be a sinusoid as the attenuation function.

Finally, we have only slightly addressed aliasing issues that occur when working on sampled data. Our algorithm can work on very low-resolution image data since the search technique efficiently smoothes blocky hard shadows. However, we expect an additional improvement of quality by using filtering schemes that also take into account the stamp size of the warped pixel or work on super-sampled depth maps.



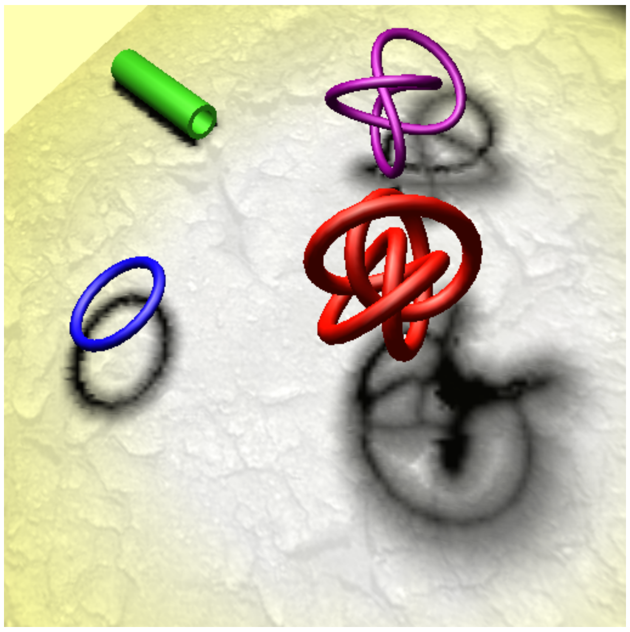


Figure 11.12: Two more complex scenes rendered with our soft shadow algorithm.

Chapter 12

Conclusions, Summary, and Future Work

12.1 Conclusions

The topic of this dissertation are shadow techniques that are suitable for interactive and real-time applications, such as computer games or virtual reality environments.

One main aspect when designing these kind of algorithms is that we need to find an implementation that corresponds to the architecture of current graphics hardware. This is because only a hardware-based method can achieve a sufficient frame rate for fully dynamic environments.

In such an environment, we can not make any assumptions about the spatial arrangement of objects and light sources. The shadow algorithm therefore has to be numerically stable and robust enough to handle all possible configurations. A software-based hierarchical scene structure additionally helps to reduce the workload by culling away large parts of the scene or pre-sorting objects to e.g. take advantage of z-culling hardware early in the pipeline. In this thesis we did not address this kind of efficiency techniques and just concentrated on the computation of shadows for the complete scene description. In principle, any algorithm that is used to cull away non-visible portions of the scene can be combined with the algorithms proposed in this thesis to reduce the workload for the shadow computation step.

Besides efficiency and numerical robustness, another important aspect of any shadow algorithm is the quality of the resulting shadows. For hard shadows, object-space techniques such as the shadow volume algorithm, provide the best visual quality since for each camera space pixel, we have exact shadow information. Image-based techniques, such as the shadow mapping technique, work on sampled data and therefore have to deal with aliasing artifacts due to the limited resolution of the shadow map.

Using extended light sources, such as linear or area lights, the computation of good-looking, accurate, and real-time soft shadows is a much more complicated task. Although the brute-force sampling based approach is to some extent possible due to the enormous gain in fill-rate and geometry performance that current graphics cards offer, this problem can only be solved efficiently by more sophisticated methods.

12.2 Summary

We now briefly summarize the shadow techniques presented in thesis, discuss their main contribution, as well as their limitations and drawbacks.

Shadow Mapping

In Part I of this thesis we focused on the shadow map technique as proposed by [Williams78]. The image-based, scene-independent nature of this algorithm along with its direct hardware support, made it a common choice for computing shadows.

In Chapter 4 we presented several extensions of Williams' original method that addressed numerical robustness and sampling quality. We showed how the distribution of depth values can be optimized by using a linear spacing and histogram equalization. In combination with the proposed depth clamping approach, we can then use most of the available depth precision for the visible parts of the scene, rather than wasting depth samples for objects outside the view frustum or large empty regions. Numerical problems during the shadow test can be avoided by storing an intermediate depth value, a hardware-based variant of Woo's [Woo92] idea. Furthermore, we presented a method of concentrating the shadow map to the visible parts of the scene. This way, we reduce aliasing artifacts since less entries in the shadow map are wasted for regions that are not visible by the current camera view. The main drawback of the methods presented in Chapter 4 is that we need to obtain some information about the actual scene. This feedback is currently provided by color-coding all relevant information and transferring it back to the CPU as an image. With future graphics hardware providing more programmability and a more general memory access model, this transfer bottleneck will no longer be an issue since all information can be processed on the card.

In Chapter 5 we focused on percentage closer filtering (PCF) for shadow maps, as proposed by Reeves et al. [Reeves87]. Although there are now some graphics cards that natively support Reeves' filtering scheme, e.g. NVIDIA's GeForce3, there are still a number of graphics systems that do not provide this special functionality. For this kind of hardware we showed that PCF is possible by extending Heidrich's color-coded shadow mapping method [Heidrich99] to multiple depth tests. In terms of numerical robustness, this color-coded approach suffers from the limited color resolution, which is about 8 to 12 bits per color channel. But on the other side, for systems, that do not support dedicated shadow map functionality or programmable pixel units, this method is the only way to implement filtered shadow maps in hardware.

Shadow maps as proposed by Williams are based on a perspective projection used when generating the shadow map. As a consequence, the shadow map has only information about those objects that are inside its field-of-view. While this perfectly fits for spot lights with a limited cut-off angle, the shadow map algorithm fails for point lights with hemispherical or omnidirectional characteristics. A solution to this was proposed in Chapter 6, where we replaced the perspective projection with a paraboloid mapping. This way, we can store a 180° view using only a single shadow map, or capture the complete environment by attaching two paraboloid shadow maps back to back. The shadow computation for an omnidirectional point light therefore can be performed in only three rendering passes. A drawback of our method is that we are forced to compute the paraboloid mapping per vertex, rather than per pixel. This requires a relatively fine tessellation of the input geometry, so that the linear interpolation of each primitive does not produce shadow artifacts.

Efficiency aspects of a shadow map implementation were the topic of Chapter 7. Here we introduced extended light maps, a special kind of light map that also includes depth information and can therefore be used just like a normal shadow map. The benefit of this approach is that most of the hardware resources are used during the generation of the shadow map. While our current implementation is based on standard 8-bit color components, we can of course improve the quality of the pre-calculated illumination and shadows by using floating point textures which are available on newer graphics cards.

Shadow Volumes

In Part II we concentrated on shadow volumes, which is the de-facto standard for computing shadows if only a small subset of the objects in the scene is dynamic. For example this is the case in many computer games, where only the game characters and some objects are moving.

In Chapter 8 we showed how to efficiently integrate the shadow volume algorithm into an interactive, hybrid-rendering system used for architectural visualization. By moving the expensive shadow calculation from the CPU to graphics hardware, we are able to dedicate more CPU resources to the computation of indirect illumination. Additionally, we store the shadow information for up to four light source in an intensity texture, which not only saves a number of rendering passes (which are very expensive for complex architectural scenes) but also allows us to include goniometric diagrams at nearly no additional cost.

A fully hardware-accelerated shadow volume implementation was presented in Chapter 9. By using programmable vertex and pixel units as well as floating point buffers, which are available on current state-of-the-art graphics cards, we are able to detect and process silhouette edges of shadow casting objects on the graphics card rather than having this step implemented on the CPU. This not only frees valuable CPU resources and speeds up the silhouette computation, but also allows mesh deformations on the hardware.

Soft Shadows

Rendering high-quality soft shadows in real-time was the focus in Part III of this thesis.

For linear light sources, we presented a novel technique that approximates soft shadows at high-quality while using only a small number of light source samples. Since soft shadow maps work on sampled data, the algorithm is to some extend independent of the scene complexity, apart from the detection of a number of occluder edges from which we generate the skin polygons. A drawback of our method is that there are situations in which the number of sample points is definitely too low such that important occluder edges are missed. Here we have to come up with a way of automatically deciding, where to add new sample points.

The soft shadow technique proposed in Chapter 11 aimed at producing visually pleasing soft shadows rather than trying to compute a physically correct solution. We have shown how to extract information about the spatial arrangement of occluder, light source, and receiver, using only a depth map obtained from one sample point. Although the algorithm at its current state produces convincing soft shadows, it still has a number of drawbacks. First, our search technique is very time consuming, since we need to perform it on the CPU. Here a hardware-based implementation would definitely help to bring the algorithm to real-time frame rates. Second, the size of the penumbra is controlled by a number of user-defined parameters that are not intuitive. Therefore we have to find a formula that e.g. is solely based on the radius of a spherical light source to which our search strategy corresponds.

12.3 Future Work

Many of the algorithms presented in this thesis focus on the fast and robust computation of hard shadows. These kinds of shadows are nowadays a common visual element in games and interactive applications. Although we are able to compute hard shadows for a number of light sources and scenes of decent complexity in real-time today, there are still a number of problems we need to solve. The shadow volume or shadow map based techniques presented in this thesis produce reasonable results at high frame rates for the polygonal models that are currently used in hardware-based rendering. But if we take a look at the scenes that are used in the field of software rendering today, we will not be able to handle these in real-time with our current techniques. As an example, rendering shadows for complex hair or fur models definitely breaks the performance of any shadow volume implementation due to the enormous geometry and fill-rate workload. Adapting software-based methods, like the hardware-based deep shadow map implementation presented by Kim and Neumann [Kim01], is often a solution for specialized applications, but can not be used efficiently in a general hardware rendering engine. The key to solve these kinds of problems is to reduce the workload to the minimum computation that is required to produce visually believing results. As an example, we can not judge if the shadow cast by one hair strand is correct or wrong, but we can definitely say if the overall shadow looks natural or not. Developing this kind of visually adaptive, hardware-accelerated shadow techniques is one of the challenging tasks for the future.

The most important direction of future research in the field of real-time shadow techniques is of course the accurate and fast computation of shadows caused by extended light sources. Although there are now a number of promising approaches, see the state-of-the-art report by [Hasenfratz03] for a good overview, there is still a lot of work to be done in this area. None of the techniques described in this report can be seen as a general-purpose, robust, and fast soft shadow technique, since too much restrictions or limitations are imposed on the shadow casting objects, receivers, or light source shapes. With future graphics hardware supporting more sophisticated programmability and data access methods, we are confident that future research will bring us to soft shadow techniques that are as general and easy to use as e.g. shadow volumes or shadow maps.

Bibliography

[Agrawala00] MANEESH AGRAWALA, RAVI RAMAMOORTHI, ALAN HEIRICH, AND LAURENT 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.

[Akenine-Möller02a] TOMAS AKENINE-MÖLLER AND ULF ASSARSSON. Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges. In P. Debevec and S. Gibson, editors, *Rendering Techniques 2002 (Eurographics Workshop Proceedings)*, pages 297–305, 2002.

[Akenine-Möller02b] TOMAS AKENINE-MÖLLER AND ERIC HAINES. *Real-Time Rendering, Second Edition*. A. K. Peters, 2002.

[Andrew79] A. M. ANDREW. Another Efficient Algorithm for Convex Hulls in Two Dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[Appel68] ARTHUR APPEL. Some Techniques for Shading Machine Renderings of Solids. In *Spring Joint Computer Conference Proceedings*, pages 37–45. AFIPS, 1968.

[Assarsson03] ULF ASSARSSON AND TOMAS AKENINE-MÖLLER. A Geometry-Based Soft Shadow Volume Algorithm Using Graphics Hardware. *ACM Transactions on Graphics*, 22(3):511–520, July 2003.

[Bergeron86] PHILIPPE BERGERON. A General Version of Crow's Shadow Volumes. *IEEE Computer Graphics & Applications*, 6(9):17–28, 1986.

[Blinn76] JAMES F. BLINN AND MARTIN E. NEWELL. Texture and Reflection in Computer Generated Images. *Communications of the ACM*, 19:542—546, 1976.

[Blinn77] JAMES F. BLINN. Models of Light Reflection For Computer Synthesized Pictures. In *Proceedings SIGGRAPH*, pages 192–198, 1977.

[Blinn88] JAMES F. BLINN. Jim Blinn's Corner: Me and my (fake) shadow. *IEEE Computer Graphics & Applications*, 8(1):82–86, January 1988.

[Blythe99] DAVID BLYTHE. Advanced Graphics Programming Techniques Using OpenGL. SIGGRAPH Course, 1999.

[Brabec00] STEFAN BRABEC AND HANS-PETER SEIDEL. Extended Light Maps. In Quasim Mehdi and Norman Gough, editors, *GAME-ON 2000 / 1st International Conference on Intelligent Games and Simulation*, pages 10–13. Society for Computer Simulation International, November 2000.

[Brabec01] STEFAN BRABEC AND HANS-PETER SEIDEL. Hardware-accelerated Rendering of Antialiased Shadows with Shadow Maps. In Horace Ho-Shing Ip, Nadia Magnenat-Thalmann, Rynson W. H. Lau, and Tat-Seng Chua, editors, *Computer Graphics International 2001 (CGI'01), July 3-6, 2001, Hong Kong, China, Proceedings*. IEEE Computer Society, 2001.

[Brabec02a] STEFAN BRABEC, THOMAS ANNEN, AND HANS-PETER SEIDEL. Practical Shadow Mapping. *Journal of Graphics Tools*, 7(4):9–18, 2002.

[Brabec02b] STEFAN BRABEC, THOMAS ANNEN, AND HANS-PETER SEIDEL. Shadow Mapping for Hemispherical and Omnidirectional Light Sources. In John Vince and Rae Earnshaw, editors, *Advances in Modelling, Animation and Rendering (Proceedings of CGI 2002)*, pages 397–408, Bradford, UK, 2002. Springer.

[Brabec02c] STEFAN BRABEC AND HANS-PETER SEIDEL. Single Sample Soft Shadows Using Depth Maps. In Michael McCool and Wolfgang Stürzlinger, editors, *Proceedings Graphics Interface*, pages 219–228, Calgary, Alberta, May 2002. Canadian Human-Computer Communications Society, A. K. Peters.

[Brabec03] STEFAN BRABEC AND HANS-PETER SEIDEL. Shadow Volumes on Programmable Graphics Hardware. *Computer Graphics Forum (Proceedings of Eurographics 2003)*, 22(3):433–440, September 2003.

[Brotman84] LYNNE S. BROTMAN AND NORMAN I. BADLER. Generating Soft Shadows with a Depth Buffer Algorithm. *IEEE Computer Graphics & Applications*, 4(10):71–81, October 1984.

[Bui-Tuong75] PHONG BUI-TUONG. Illumination for Computer Generated Images. *Communications of the ACM*, 18(6):311–317, June 1975.

[Carmack00] JOHN CARMACK. John Carmack on shadow volumes. Available from `http://www.nvidia.com`, May 2000.

[Catmull75] EDWIN CATMULL. Computer Display of Curved Surfaces. In *Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures*, pages 11–17, May 1975.

[Chen93] SHENCHANG ERIC CHEN AND LANCE WILLIAMS. View interpolation for image synthesis. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 279–288, 1993.

[Chin89] NORMAN CHIN AND STEVEN FEINER. Near Real-Time Shadow Generation Using BSP Trees. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 99–106, July 1989.

[Chrysanthou95] YIORGOS CHRYSANTHOU AND MEL SLATER. Shadow Volume BSP Trees for Computation of Shadows in Dynamic Scenes. *1995 Symposium on Interactive 3D Graphics*, pages 45–50, April 1995.

[Cook84] ROBERT L. COOK, THOMAS PORTER, AND LOREN CARPENTER. Distributed Ray Tracing. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 137–145, July 1984.

[Crow77] FRANKLIN C. CROW. Shadow algorithms for computer graphics. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, volume 11, pages 242–248, July 1977.

[Diefenbach94] PAUL J. DIEFENBACH AND NORMAN BADLER. Pipeline Rendering: Interactive Refractions, Reflections and Shadows. *Displays: Special Issue on Interactive Computer Graphics*, 15(3):173–180, 1994.

[Dietrich01] SIM DIETRICH. *Shadow Techniques*. NVIDIA Corporation, 2001. Available from `http://www.nvidia.com`.

[Dmitriev02] KIRILL DMITRIEV, STEFAN BRABEC, KAROL MYSZKOWSKI, AND HANS-PETER SEIDEL. Interactive Global Illumination Using Selective Photon Tracing. In Paul Debevec and Simon Gibson, editors, *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 21–33, Pisa, Italy, 2002. Eurographics/ACM.

[Drettakis94] GEORGE DRETTAKIS AND EUGENE FIUME. A Fast Shadow Algorithm for Area Light Sources Using Backprojection. In *"Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 223–230, July 1994.

[Everitt02] CASS EVERITT AND MARK J. KILGARD. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. Technical report, NVIDIA Cooperation, March 2002. Available from `http://www.nvidia.com`.

[Fernando01] RANDIMA FERNANDO, SEBASTIAN FERNANDEZ, KAVITA BALA, AND DONALD P. GREENBERG. Adaptive Shadow Maps. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 387–390, August 2001.

[Foley96] JAMES D. FOLEY, ANDRIES VAN DAM, STEVEN K. FEINER, AND JOHN F. HUGHES. *Computer Graphics — Principles and Practice*. The Systems Programming Series. Addison-Wesley, second edition, 1996.

[Glassner95] ANDREW GLASSNER. *Principles of Digital Image Synthesis*. Morgan Kaufmann, 1995.

[Gonzalez92] RAFAEL GONZALEZ AND RICHARD WOODS. *Digital Image Processing*. Addison-Wesley, 1992.

[Gooch99] BRUCE GOOCH, PETER-PIKE J. SLOAN, AMY GOOCH, PETER S. SHIRLEY, AND RICH RIESENFELD. Interactive Technical Illustration. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 31–38. ACM SIGGRAPH, April 1999.

[Govindaraju03] NAGA GOVINDARAJU, BRANDON LLOYD, SUNG-EUI YOON, AVNEESH SUD, AND DINESH MANOCHA. Interactive Shadow Generation in Complex Environments. Technical report, University of North Carolina, 2003.

[Haeberli90] PAUL E. HAEBERLI AND KURT AKELEY. The Accumulation Buffer: Hardware Support for High-Quality Rendering. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, volume 24, pages 309–318, August 1990.

[Haeberli93] PAUL E. HAEBERLI AND MARK SEGAL. Texture Mapping As A Fundamental Drawing Primitive. *Fourth Eurographics Workshop on Rendering*, pages 259–266, June 1993.

[Haines01] ERIC HAINES. Soft planar shadows using plateaus. *Journal of Graphic Tools*, 6(1):19–27, 2001.

[Hasenfratz03] JEAN-MARC HASENFRATZ, MARC LAPIERRE, NICOLAS HOLZSCHUCH, AND FRANÇOIS SILLION. A survey of Real-Time Soft Shadows Algorithms. In *Eurographics*. Eurographics, Eurographics, 2003. State-of-the-Art Report.

[Heckbert92] PAUL HECKBERT. Discontinuity Meshing for Radiosity. In *Rendering Techniques '92 (Proc. of Eurographics Rendering Workshop)*, pages 203–226, May 1992.

[Heckbert97] PAUL HECKBERT AND MICHAEL HERF. Simulating Soft Shadows with Graphics Hardware. Technical Report CMU-CS-97-104, Carnegie Mellon University, January 1997.

[Heidmann91] TIM HEIDMANN. Real Shadows Real Time. *IRIS Universe*, 18:28–31, 1991.

[Heidrich98a] WOLFGANG HEIDRICH, JAN KAUTZ, PHILIPP SLUSALLEK, AND HANS-PETER SEIDEL. Canned Lightsources. In *Eurographics Rendering Workshop 1998*, pages 293–300, June 1998.

[Heidrich98b] WOLFGANG HEIDRICH AND HANS-PETER SEIDEL. View-independent environment maps. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 39–46, August 1998.

[Heidrich99] WOLFGANG HEIDRICH. *High-quality Shading and Lighting for Hardware-accelerated Rendering*. PhD thesis, University of Erlangen, Computer Graphics Group, 1999.

[Heidrich00] WOLFGANG HEIDRICH, STEFAN BRABEC, AND HANS-PETER SEIDEL. Soft Shadow Maps for Linear Lights.

Rendering Techniques 2000: 11th Eurographics Workshop on Rendering, pages 269–280, June 2000.

[Hourcade85] JEAN-CHARLES HOURCADE AND ALAIN NICOLAS. Algorithms for Antialiased Cast Shadows. *Computers & Graphics*, 9(3):259–265, 1985.

[Isard02] MICHAEL ISARD, MARK SHAND, AND ALAN HEIRICH. Distributed rendering of interactive soft shadows. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 71–76. Eurographics Association, 2002.

[Kautz03] JAN KAUTZ. *Realistic, Real-Time Shading and Rendering of Objects with Complex Materials*. PhD thesis, Max-Planck-Institut für Informatik, 2003.

[Keating99] BRETT KEATING AND NELSON MAX. Shadow Penumbras for Complex Objects by Depth-Dependent Filtering of Multi-Layer Depth Images. In *Rendering Techniques '99 (Proc. of Eurographics Rendering Workshop)*, pages 197–212, June 1999.

[Kessenich03] JOHN KESSENICH, DAVE BALDWIN, AND RANDI ROST. The OpenGL Shading Language, February 2003. draft version 1.05.

[Kilgard00] MARK J. KILGARD. Shadow Mapping with Today's OpenGL Hardware, March 2000. Available from `http://www.nvidia.com`.

[Kim01] TAE-YONG KIM AND ULRICH NEUMANN. Opacity Shadow Maps. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pages 177–182, June 2001.

[Kirsch03] FLORIAN KIRSCH AND JÜRGEN DÖLLNER. Real-time soft shadows using a single light sample. *Journal of WSCG*, 11(1), 2003.

[Lengyel02] ERIC LENGYEL. The Mechanics of Robust Stencil Shadows. Tutorial available on www.gamasutra.com, Oct 2002.

[Leonardo da Vinci45] LEONARDO DA VINCI. *The notebooks of Leonardo da Vinci*. Cape, 1945. arranged, rendered into English and introd. by Edward MacCurdy.

[Lindholm01] ERIK LINDHOLM, MARK J. KILGARD, AND HENRY MORETON. A User-Programmable Vertex Engine. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 149–158, August 2001.

[Lokovic00] TOM LOKOVIC AND ERIC VEACH. Deep Shadow Maps. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 385–392, July 2000.

[Loscos97] CÉLINE LOSCOS AND GEORGE DRETTAKIS. Interactive High-Quality Soft Shadows in Scenes with Moving Objects. In *Eurographics '97*, Jun 1997.

[Mark97] WILLIAM MARK, LEONARD MCMILLAN, AND GARY BISHOP. Post-Rendering 3D Warping. In *Proceedings of the Symposium on Interactive 3D Graphics*, pages 7–16, April 1997.

[Mark03] WILLIAM R. MARK, R. STEVEN GLANVILLE, KURT AKELEY, AND MARK J. KILGARD. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Transactions on Graphics*, 22(3):896–907, July 2003.

[McCool00] MICHAEL D. MCCOOL. Shadow volume reconstruction from depth maps. *ACM Transactions on Graphics*, 19(1):1–26, January 2000.

[Microsoft00] MICROSOFT. *DirectX SDK Documentation*. Microsoft Developer Network, 2000. Available from `http://msdn.microsoft.com/directx`.

[Newell72] MARTIN E. NEWELL, RICHARD G. NEWELL, AND TOM L. SANCHA. A solution to the hidden surface problem. In *Proceedings of the ACM Annual Conference*, pages 443–450, 1972.

[Nishita85] TOMOYUKI NISHITA, ISAO OKAMURA, AND EIHACHIRO NAKAMAE. Shading Models for Point and Linear Sources. *ACM Transactions on Graphics*, 4(2):124–146, April 1985.

[NVIDIA99] NVIDIA. *Perfect Reflections and Specular Lighting Effects With Cube Environment Mapping*, 1999. Available from `http://www.nvidia.com`.

[NVIDIA02] NVIDIA. *OpenGL Extension Specifications*, 2002. Available from `http://www.nvidia.com`.

[Ouellette99a] Marc Ouellette and Eugene Fiume. Approximating the Location of lntegrand Discontinuities for Penumbral Illumination With Linear Light Sources. In *Graphics Interface '99*, pages 66–75, June 1999.

[Ouellette99b] Marc Ouellette and Eugene L. Fiume. Approximating the Location of Integrand Discontinuities for Penumbral Illumination Computation with Area Light Sources. In *Eurographics Rendering Workshop 1999*, June 1999.

[Parker98] Steven Parker, Peter Shirley, and Brian Smits. Single Sample Soft Shadows. Technical Report UUCS-98-019, Computer Science Department, University of Utah, 1998.

[Pirzadeh99] Hormoz Pirzadeh. Computational geometry with the rotating calipers. Master's thesis, School of Computer Science, McGill University, Montreal, Quebec, Canada, November 1999.

[Pop01] Mihai Pop, Christian Duncan, Gill Barequet, Michael Goodrich, Wenjing Huang, and Subodh Kumar. Efficient perspective-accurate silhouette computation and applications. In *Proceedings of the seventeenth annual symposium on Computational geometry*, pages 60–68. ACM Press, 2001.

[Purcell02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[Reeves87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering Antialiased Shadows with Depth Maps. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 283–291, July 1987.

[Roettger02] Stefan Roettger, Alexander Irion, and Thomas Ertl. Shadow Volumes Revisited. In V. Skala, editor, *Proceedings of WSCG 2002*, pages 373–379, 2002.

[Schmittler02] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR – A Hardware Architecture For Ray Tracing. In *Proceedings of the conference on Graphics Hardware 2002*, pages 27–36. Saarland University, Eurographics Association, 2002. Available from `http://www.openrt.de`.

[Segal92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul E. Haeberli. Fast shadows and lighting effects using texture mapping. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 249–252, July 1992.

[Segal98] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2)*, 1998.

[Soler98] Cyril Soler and François X. Sillion. Fast Calculation of Soft Shadow Textures Using Convolution. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 321–332, July 1998.

[Stamminger02] Marc Stamminger and George Drettakis. Perspective Shadow Maps. *ACM Transactions on Graphics*, 21(3):557–562, July 2002.

[Stewart94] A. James Stewart and Sherif Ghali. Fast Computation of Shadow Boundaries Using Spatial Coherence and Backprojections. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 231–238, July 1994.

[Sutherland74] Evan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *ACM Computing Surveys (CSUR)*, 6(1):1–55, 1974.

[Toussaint83] Godfried T. Toussaint. Solving geometric problems with the rotating calipers. In *Proceedings of IEEE MELECON'83, Athens, Greece*, May 1983.

[Udeshi99] Tushar Udeshi and Charles Hansen. Towards interactive, photorealistic rendering of indoor scenes: A hybrid approach. In *Rendering Techniques '99 (Proc. of Eurographics Rendering Workshop)*, pages 63–76, June 1999.

[Voorhies94]	DOUGLAS VOORHIES AND JIM FORAN. Reflection Vector Shading Hardware. *Proceedings of SIGGRAPH 94*, pages 163–166, July 1994.
[Wald01]	INGO WALD, PHILIPP SLUSALLEK, AND CARSTEN BENTHIN. Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pages 277–288, June 2001.
[Whitted80]	TURNER WHITTED. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, June 1980.
[Williams78]	LANCE WILLIAMS. Casting Curved Shadows on Curved Surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, volume 12, pages 270–274, August 1978.
[Woo90]	ANDREW WOO, PIERRE POULIN, AND ALAIN FOURNIER. A Survey of Shadow Algorithms. *IEEE Computer Graphics & Applications*, 10(6):13–32, November 1990.
[Woo92]	ANDREW WOO. *Graphics Gems III*, chapter The Shadow Depth Map Revisited, pages 338–342. Academic Press, 1992.
[Woo99]	MASON WOO, JACKIE NEIDER, TOM DAVIS, AND DAVE SHREINER. *OpenGL Programming Guide, Third Edition*. Addison-Wesley, 1999.
[Wyman03]	CHRIS WYMAN AND CHARLES HANSEN. Penumbra Maps. Technical report, University of Utah, School of Computing, April 2003.
[Ying02]	ZHENGMING YING, MIN TANG, AND JINXIANG DONG. Soft Shadow Maps for Area Light by Area Approximation. In *10th Pacific Conference on Computer Graphics and Applications*, pages 442–443. IEEE, 2002.
[Zhang98]	HANSONG ZHANG. Forward Shadow Mapping. In *Eurographics Rendering Workshop 1998*, pages 131–138, June 1998.