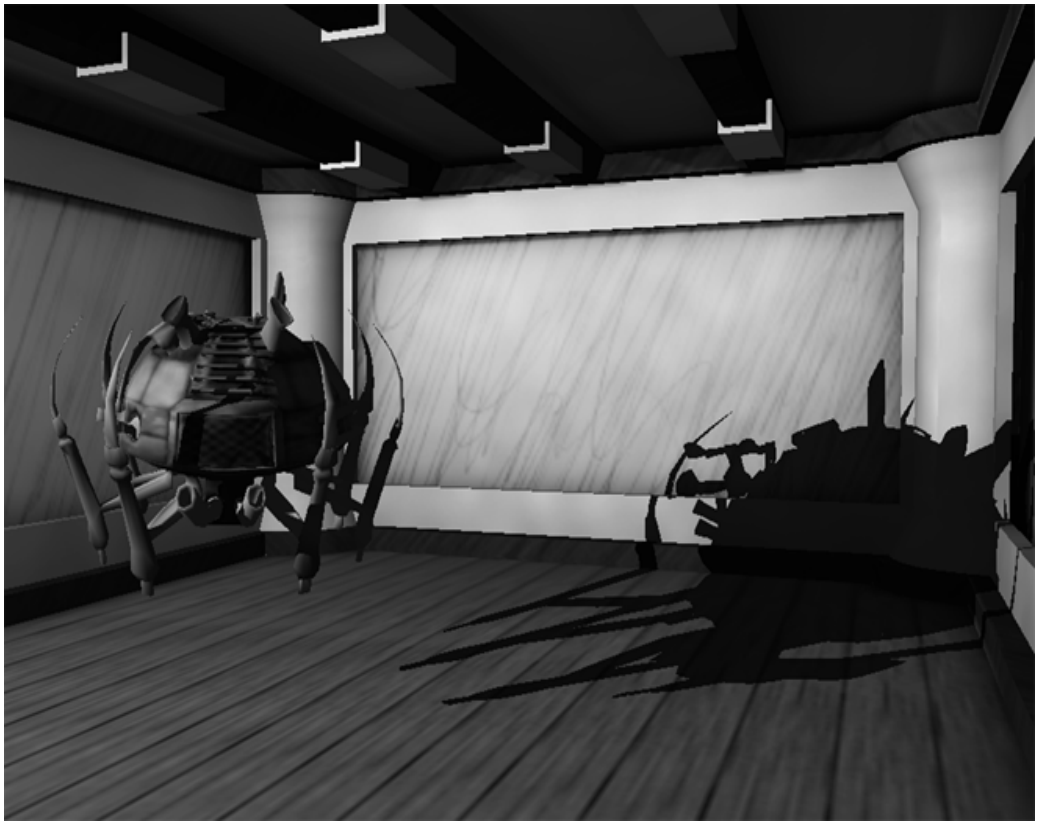
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

An Empirically Derived System for High-Speed Shadow Rendering

by Pierre (HR) Rautenbach



Submitted in partial fulfilment of the requirements for the degree

Magister Scientiae (Computer Science)

in the Faculty of Engineering, Built-Environment

and Information Technology

University of Pretoria

Pretoria

November 21st 2008

Abstract

Shadows have captivated humanity since the dawn of time; with the current age being no exception – shadows are core to realism and ambience, be it to invoke a classic Baroque interplay of lights, darks and colours as the case in Rembrandt van Rijn's *Militia Company of Captain Frans Banning Cocq* or to create a sense of mystery as found in film noir and expressionist cinematography. Shadows, in this traditional sense, are regions of blocked light – the combined effect of placing an object between a light source and surface.

This dissertation focuses on real-time shadow generation as a subset of 3D computer graphics. Its main focus is the critical analysis of numerous real-time shadow rendering algorithms and the construction of an empirically derived system for the high-speed rendering of shadows. This critical analysis allows us to assess the relationship between shadow rendering quality and performance. It also allows for the isolation of key algorithmic weaknesses and possible bottleneck areas. Focusing on these bottleneck areas, we investigate several possibilities of improving the performance and quality of shadow rendering; both on a hardware and software level. Primary performance benefits are seen through effective culling, clipping, the use of hardware extensions and by managing the polygonal complexity and silhouette detection of shadow casting meshes. Additional performance gains are achieved by combining the depth-fail stencil shadow volume algorithm with dynamic spatial subdivision.

Using this performance data gathered during the analysis of various shadow rendering algorithms, we are able to define a fuzzy logic-based expert system to control the real-time selection of shadow rendering algorithms based on environmental conditions. This system ensures the following: nearby shadows are always of high-quality, distant shadows are, under certain conditions, rendered at a lower quality and the frames per second rendering performance is always maximised.

Key words and phrases: Shadow Algorithms, Spatial Subdivision, Stencil Shadow Volumes, Shadow Mapping, Instruction Set Utilisation, Expert Systems, Fuzzy Logic.

Supervisors: Prof. D.G. Kourie, V. Pieterse
Department of Computer Science
Degree: Magister Scientiae

Acknowledgements

The author would like to thank the following:

- My gratitude and eternal thanks to Professor Derrick Kourie and Mrs. Vreda Pieterse for their constant guidance, patience, support, encouragement and inspiration throughout production of this work.

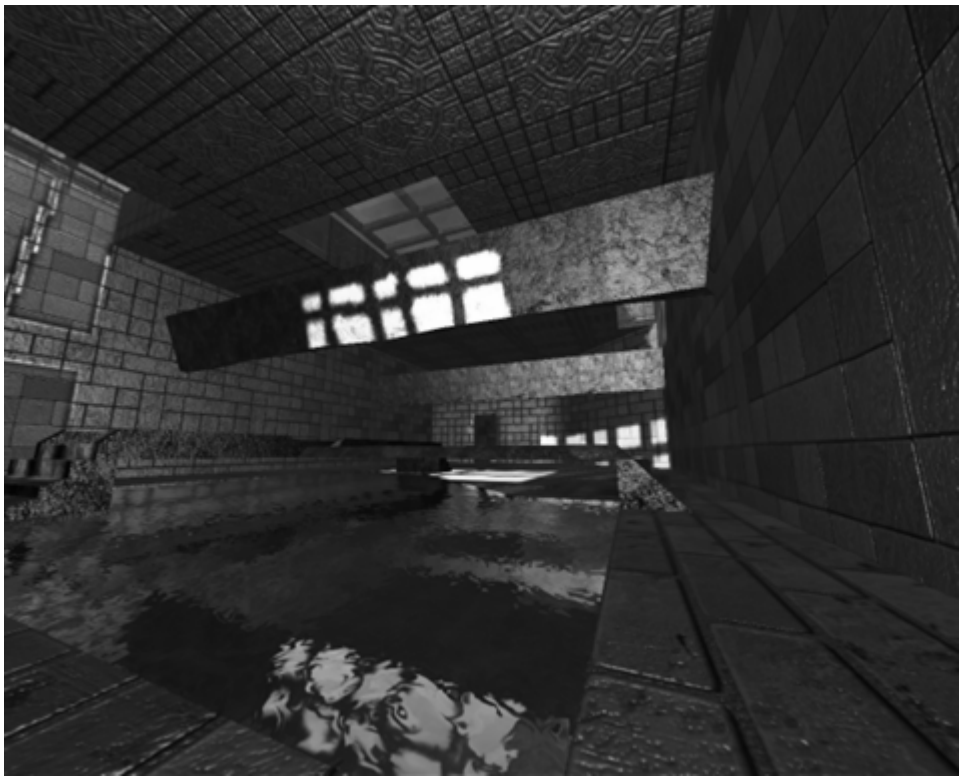
- My endless respect to industry pioneers like Mark Kilgard (NVIDIA), John Carmack (id Software), Randy Fernando (NVIDIA) and Tim Sweeney (Epic Games) for everlasting inspiration, groundbreaking research and ceaseless innovation.

- Special thanks to some of the University of Pretoria's best students; Nelis Franken, Michael Kohn, Chris Schulz, Jaco Prinsloo, Morkel Theunissen and Will van Heerden. Both as friends and colleagues they have made life at the university a lot more enjoyable.

- Last but not least, special thanks to my family and friends for their love and encouragement, especially my mom, Wilna, and dad, Ig, for their incessant motivation and support through the years.

- *The financial assistance of the National Research Foundation towards this research is hereby acknowledged. Opinions expressed in this thesis and conclusions arrived at, are those of the author and not necessarily to be attributed to the National Research Foundation.*

Table of Contents



Chapter 1

Introduction

Chapter 1 presents the general research domain, research problem and overall dissertation structure.

In this chapter we will introduce:

- The research domain
- The research problem
- A general outline of the work addressing the problem

1.1 Research Domain

In order to contextualise shadow rendering within its historical context, this chapter starts by offering a brief overview of computer gaming – the primary driving force behind boundary defying real-time rendering algorithms such as those developed for shadow generation. Also, please note, portions of this section are sourced from the author's textbook, *3D Game Programming Using DirectX 10 and OpenGL* (Rautenbach, 2008).

The first computer game ever was a crude noughts and crosses simulation written in 1952 (Winter, 2004). This game, called *OXO*, was developed by Sandy Douglas using an EDSAC computer (one of the first stored program electronic computers). The user used a rotary telephone dial for input with the output being generated on a 35 by 16 pixel cathode ray tube display (Campbell-Kelly, 2006). Figure 1.1 shows an emulation of the original program.

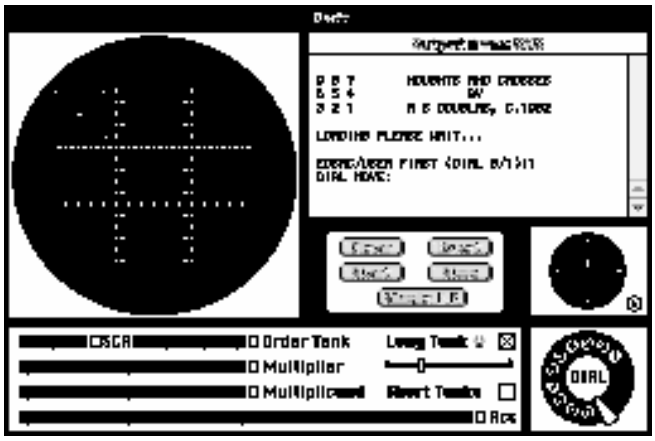


Figure 1.1 A screenshot of the game OXO.

William Higinbotham, an American physicist, created *Tennis for Two* in 1958 using an oscilloscope (OSTI, 1981). This game showed a side view of a tennis court and the player was required to hit a gravity affected ball over a net. Tennis for Two is considered by many as the first computer game due to the EDSAC computer being mainly limited to the University of Cambridge Mathematical Laboratory in England. Figure 1.2 shows Tennis for Two running on an oscilloscope.

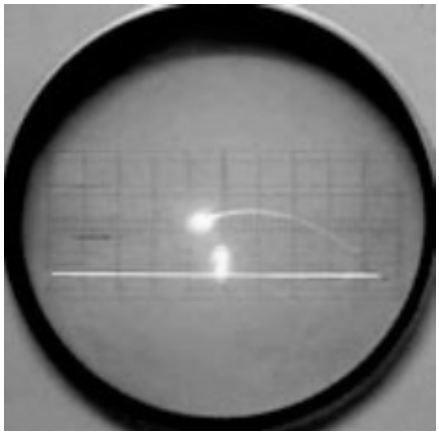


Figure 1.2 A photograph of the game Tennis for Two.

The 1960s saw the advent of computer gaming on mainframe computers. Most of these games were text-based adventures with MUDs (Multi-User Dungeons) appearing in the late 1970s (Klietz, 1992). These MUDs, existing to this very day, were some of the first networked games, with the original MUDs requiring a connection to an academic network. A *MUD* typically combines elements of role-playing and chat room style social interaction. All actions and dialog in the environment are text driven. Modern MMOGs such as *World of Warcraft*, *Guildwars* and *Dungeons & Dragons Online* have several similarities to early MUDs and can loosely be considered as graphical next-generation MUDs.

PONG, designed by Nolan Busnell, led to the birth of Atari Interactive and was mainly distributed via coin-operated arcade machines and home consoles (Miller, 2005). The original PONG was related to Higinbotham's Tennis for Two, with the exception of being based on the sport of table tennis and for having a top down view. PONG made use of solid lines to represent paddles, a dotted line to represent the net and a square to represent the ball. Many versions of the original Atari classic have been made over the years and the entire genre of ball-and-bat video games have become known as *Pong* games (note the lower case spelling). Figure 1.3 shows a clone of the original classic using DirectDraw.

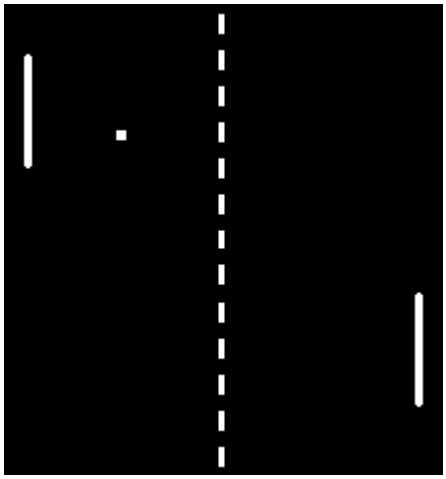


Figure 1.3 A PONG clone.

The Atari 2600 (Figure 1.4), released in 1977, allowed for the use of plug-in cartridges (Yarusso, 2007). Dedicated consoles offering one or two games were the norm before then and having one console supporting a theoretically unlimited number of games, such as *Breakout*, *Donkey Kong*, *Pac-Man* and *Space Invaders*, was extremely popular with the buying market and contributed heavily towards the growth of computer gaming.

Figure 1.4 The Atari 2600.

The term *personal computer game* or *PC game* surfaced with the release of the Apple II (see Figure 1.5) in 1977 (Weyhrich, 2002). Although the Apple II offered some productivity and business applications such as a spreadsheet and word processor, it was designed specifically with educational and personal use in mind. The Apple II was shipped with two well-documented and easy to learn BASIC programming languages, *Applesoft* and *Integer*, resulting in the Apple II being used by many computer enthusiasts learning how to program. Applesoft BASIC, created by Microsoft, supported floating point arithmetic and was initially offered as an upgrade to Integer BASIC and later included with the release of the *Apple II Plus*. The Apple II enjoyed a phenomenal user base and grew into the most popular game development platform of the time with hundreds of titles shipped. Two of the world's most respected and prolific game developers, *John Romero* and *John Carmack* (responsible for genre-defining games such as *Doom* and *Quake*), started their careers programming games for the Apple II (Kushner, 2003:23-24,33-37,41).

Figure 1.5 One of the first Apple II computers.

The 1980s saw the advent of the IBM PC (and compatibles), *Commodore 64*, *Atari ST*, etc (Reimer, 2005). The general idea behind all these systems was 'a personal computer for the masses'. The original IBM PCs of the early 1980s were priced out of the reach of most home users (an example is shown in Figure 1.6) but gained significant market share in the business sector. IBM PCs featured Microsoft BASIC as programming language and an open architecture allowing other manufacturers to develop both peripherals and software for it. This open architecture is the primary reason for the popularity of the PC today. The Commodore 64 featured impressive

graphics and sound capabilities compared to the Apple II and IBM PCs of the time. It was also priced much more aggressively than its counterparts. The Commodore 64 also competed against video game consoles such as the Atari 2600 by allowing direct connectivity with a television set. The 'video game crash of 1983' led to the bankruptcy of numerous video game, console and home computer manufacturers (Taylor, 1982). This industry crash was the direct result of the video game market being swamped by a large number of sub-quality games and the availability of competitively priced personal computer systems fulfilling multiple educational, business and entertainment roles. With video game console companies collapsing, PC games quickly took the place of their console counterparts.

Figure 1.6 The IBM PC Junior released in 1983.

The Atari ST (see Figure 1-7) was released in 1985 and was especially suited for PC gaming due to its colourful graphics, good sound, fast performance and good price (Powell, 1985). 3D computer games such as *Dungeon Master* and notable classics such as *Peter Molyneux's Populous* (also released on the PC and various other platforms) were created for it. The PC, although lagging behind at the beginning of the 1980s, slowly gained popularity due to its open architecture, dropping price, easy upgrading and usefulness as a business tool. The IBM PC compatible was at the forefront of the personal computer race at the start of the 1990s, and the release of Windows 3.0 in May 1990 in particular led to the PC becoming the computing platform of choice to this very day.

Figure 1.7 The Atari ST computer.

The introduction of high quality soundcards, high resolution displays and peripherals such as the computer mouse and joystick really drove the adoption of computer gaming but it wasn't until 1992 that the real power of the PC as a gaming platform was realised. The main game responsible for this was *id Software*'s shareware mega-hit Wolfenstein 3D. Wolfenstein 3D popularized the first-person shooter genre and the PC as a gaming platform by allowing the player to interact with a virtual environment from a first-person perspective. Wolfenstein 3D was of course not the first 3D computer game for the PC with id Software employing and refining the technology that would become Wolfenstein 3D in *Hovertank 3D* and *Catacomb 3D* during 1991. Other older PC games such as *Elite* also featured 3D environments but never achieved the level of technical complexity of Wolfenstein 3D nor its cultural and industry impact. Another breakthrough in the graphics of 3D games came with id Software's release of Doom in 1993. Doom, a screenshot of which is shown in Figure 1.8, really revolutionised the gaming industry (GameSpy, 2001) with its fast paced network play and immersive graphics and companies like Microsoft started spending millions of dollars on research and development to migrate gaming from MS-DOS to their Windows platform (Craddock, 2007). This research and development culminated in the DirectX Application Programming Interface (API).

Figure 1.9 id Software's Doom released in 1993.

Following the release of Doom, Microsoft wanted to establish *Windows 95* as the gaming platform of choice, as opposed to MS-DOS still being used by the majority of

games throughout 1995 and 1996. During a Microsoft Halloween media event at the end of 1995, called *Judgement Day*, a 32-bit port of Doom was showcased featuring a video address by *Bill Gates* superimposed inside the game proclaiming Windows 95, using the DirectX API, as “thee game platform” (Microsoft, 1995). Initial DirectX versions were not unequivocally successful products but were nonetheless important as technological building blocks. Most of the issues associated with these initial DirectX releases were, however, resolved with the release of DirectX 5.0 in 1997 and the era of MS-DOS based games was officially over. There were also a number of developers using OpenGL due to it being a cross-platform graphics API unlike Microsoft’s Direct3D. OpenGL has since had a strong footing in the science and gaming’s first-person shooter genre, not only because of its cross-platform nature but also much due to its minimalist design as opposed to Direct3D’s perceived complexity. Direct3D’s (DirectX’s graphics library) inception and the standardisation of its competitor, OpenGL, together with the advent of mainstream 3D accelerated graphics hardware revolutionised computer gaming and led to a new era of ever more realistic 3D graphics and constant improvements in graphics hardware. The first-person shooter is generally considered the primary benchmark for graphics complexity, realism and visual effects with *Doom3* and the *Quake, Unreal and Half-Life* series often setting the standard for other titles.

The progression of Direct3D and OpenGL is closely coupled with the development of 3D accelerated graphic cards. These libraries can be considered a series of specifications that requires implementation by graphic hardware vendors. Hardware support enables the rapid execution of graphics calls, functions, or effects – in the process freeing the CPU to do other calculations. The GPU (graphics processing unit), integrated into a video card, is a dedicated graphics rendering device and controls the rendering quality and drawing performance depending on the number of supported specifications. The first mainstream GPUs were released with the Atari ST, the Commodore Amiga and some home computers of the 1980s (Knight, 2003). These GPUs were nothing more than simple *blitters* responsible for moving bitmaps around in memory. In 1991 S3 Graphics launched the first mainstream 2-D accelerator for the PC and was soon followed by 2-D accelerators with added 3-D features such as the *ATI Rage* and the *S3 ViRGE* (Bell, 2003). These basic graphics accelerators soon evolved to include support for *transform and lighting* (translating three-dimensional objects and calculating the effects of lighting on objects) with the release of DirectX 5.0 and progressed to include programmable shaders in addition to numerous other advancements with later releases of DirectX and OpenGL.

Computer gaming today is a multi-billion dollar industry with 2004’s U.S retail stales set at more $9.9 billion. According to the NPD Group, there was a 4% increase in unit sales as compared to the previous year. This highly-profitable situation is playing itself out throughout the world. A report released by iResearch predicts China's online game revenue to reach $970 million in 2008 – a 28% increase from last year. According to the IDSA more than 60% of Americans age six and older (145 million people) play computer and video games with the average game player being 28 years old. With the demand for new titles a constant factor and the number of emerging developers always increasing, the market for games with boundary defying

graphics are set to increase for quite some time to come – for example, *Grand Theft Auto IV* broke sales records by selling about 3.6 million units on its first day of release and grossing more than $500 million in its first week. As of 16 August 2008, the game has sold over 10 million copies (Ortutay, 2008).

This ever constant push for "immersive and more realistic" computer games has resulted in countless innovations over the years – the early 90s seeing the use of spatial subdivision and multi-texturing techniques with games released in the mid-2000s becoming known for their use of real-time shadows and advanced shader techniques. A good example of such a game is id Software's *Doom3* which specifically utilised stencil shadow volumes to add not only realism but also suspense and atmosphere (Carmack, 2000). The problem with shadows is, simply put, performance. Doom3, released in 2003, required high-end hardware to run as intended; that said, the player had the option of deactivating performance compromising elements such as shadows and specularity. However, disabling these features resulted in a less than satisfactory gaming experience. Shadows and other special effects such as specular highlights and real-time reflections have become expected, and today's mid-range hardware is more than adequate in handling each of these effects separately. However, the performance impact remains an issue when real-time shadows are coupled with AI sub-routines such as cognitive model based Non-Player Character (NPC) interaction, input control, shader effects such as reflective water, motion blur and specular bump mapping, 3D spatialisation and material based distortion for sound, realistic object interaction based on Newton's Laws, etc.

Mobile devices such as the iPhone also represent a vast untapped market for game development and graphical applications (Apple's projections estimate over 20 million iPhones shipped by the end of 2008). The iPhone, as a mass mobile platform, features powerful hardware, display and input technology – technology presenting the user with a realistic gaming experience. The iPhone and iPod Touch have the potential of not just cutting into the mobile gaming market, but to actually dethrone the Sony PSP and Nintendo DS. With the iPhone SDK being the most talked about release in recent times, there is already enormous interest to create applications and especially games targeting this platform. This unparalleled interest has resulted in more than 500 applications (with 241 in the game category) being available on the same day as Apple's delivery platform, the AppStore's, launch. Games targeting this platform have an even harder time when it comes to performance balancing, for example, the iPhone features a 620 MHz ARM 1176 CPU underclocked to 412 MHz with it's Graphical Processing Unit (offering support for OpenGL ES) being a PowerVR MBX Lite 3D unit (Apple, 2008). Running high-quality immersive games on the iPhone is thus a classic example of the need for performance balancing, especially when rendering shadows and other advanced special effects.

Research resulting in numerous shadow rendering algorithms has been conducted since the late 1960s and has picked up great momentum with the evolution of high-end dedicated graphics hardware. Some of these algorithms, like shadow mapping and shadow volumes, are more successful than others. The success of an algorithm

is dependent on the balance between speed and realism and techniques like shadow mapping and stencil shadow volumes are particularly amenable to hardware implementation – thus freeing the CPU of a substantial processing burden and making the real-time rendering of shadows feasible (Kilgard, 1999).

The study of real-time shadow rendering is not new, and has involved numerous industry leading researchers as well as professional practitioners in the past. The most recent work includes techniques such as variance shadow mapping (Lauritzen, 2006) and cascaded shadow maps (Dimitrov, 2007). The former solves the problem of shadow map aliasing (with minimal additional storage and computation) with the latter being extremely amenable to shadows cast over large terrain-based areas. These techniques extend the traditional shadow mapping algorithm introduced by Williams (1978); they are also less sensitive to geometric complexity when compared to the basic stencil shadow volume algorithm (Kilgard, 1999). The only problem is their environmental dependencies. Cascaded shadow maps (Dimitrov, 2007) will, for example, only be effective when large outdoor environments are to be shadowed with variance shadow mapping (Lauritzen, 2006) reducing aliasing artefacts at the cost of “light bleeding”.

Several shadowing algorithms, including the fundamentals and implementation of shadow volumes and shadow mapping are investigated in subsequent chapters – a brief summary is given here:

- A quite complex, and now mostly redundant shadow algorithm was introduced by Appel (1968) and further developed by Bouknight and Kelley (1970). This algorithm, commonly known as scan line polygon projection, adds shadow generation to scan-line rendering.

- An extremely easy to use shadow generation technique was described by Blinn (1988). This method simply calculates the projection of an object on some base-plane.

- Lance Williams introduced the concept of shadow mapping in 1978. His primary aim was the rendering of shadows on curved surfaces. Shadow mapping adds shadows to a scene by testing whether a particular pixel is hidden from a light source.

- The original shadow volume concept was introduced by Frank Crow in 1977. Crow defined a shadow volume as three-dimensional area occluding objects and surfaces from a light source. This original approach has since been extended to incorporate the generation of soft-edged shadows, including revision of the algorithm to utilise modern-day 3-D acceleration capabilities.

- The first feasible real-time shadow volume algorithm was introduced by Tim Heidmann in 1991. His algorithm, building on Crow’s work, makes use of the 3-D accelerator’s stencil buffer – effectively limiting the render area (through a process called stencilling).

- Michael McCool (2000) proposed shadow volume reconstruction using depth maps. His algorithm has a slight performance advantage over the traditional approach without any loss in shadow quality.

- Soft-edged shadows using penumbra wedges was proposed by Akenine-Möller and Assarsson (2002).

- Thakur et al (2003) developed a discrete algorithm for improving the Heidmann original. Their algorithm is primarily based on the elimination of various testing phases.

- Another interesting hybrid approach is the one developed by Eric Chan and Frédo Durand (2004). Their approach combines the strengths of shadow maps and shadow volumes to produce a hybrid algorithm for the efficient rendering of pixel-accurate hard-edged shadows.

- Kolic et al (2004), in turn, developed a shadowing technique purely focussing on the utilisation of current GPU advances. Their algorithm specifically deals with the casting of shadows on concave complex objects such as trees.

- Rautenbach et al (2008) combined the depth-fail stencil shadow volume algorithm with Octree-based spatial subdivision to improve the rendering performance of polygonal environments with static light sources.

Several other shadow algorithms were also originally considered for inclusion. However, the previously listed ones were chosen due to their overall spectrum of coverage; that is, their utilisation of GPU advances and distinct improvements and/or approach to the shadow rendering problem. Other approaches originally considered include: Stefan Brabec and Hans-Peter Deidel's (2002) single sample soft shadows using depth maps, George Drettakis and Eugene Fiume's (1994) fast shadow algorithm for area light sources using back-projection, Randima Fernando et al's (2001) adaptive shadow mapping approach, Eric Haines' (2001) soft planar shadows using plateaus, Wolfgang Heidrich et al's (2000) soft shadow maps for linear lights, Hourcade and Nicolas' (1985) algorithms for antialiased cast shadows, Daniel Kersten et al's (1994, 1997) approach to moving cast shadows, Florian Kirsch and Juergen Doellner's (2003) real-time soft shadows using a single light sample, Tom Lokovic and Eric Veach's (2000) deep shadow maps, William T. Reeves et al's (1987) antialiased shadows with depth maps algorithm and Mark Segal et al's (1992) fast shadows and lighting effects using texture mapping.

1.2 Problem Statement

The fast evolving computer gaming industry is governed by a constant need for increased realism and total immersion (with the need for increased realism being addressed by a number of shader techniques such as reflections, refraction,

specularity and shadows). The presented research seeks to offer a state of the art review of existing shadow rendering algorithms. The aim of this review is to describe numerous issues related to real time shadow rendering, to identify gaps and to suggest improvements to known algorithms as well as to discover directions for the development of new ones.

When considering the existing algorithms, it can easily be observed that shadow rendering solutions are implementation/requirement specific. Woo (1990) points out that knowledge related to algorithms can be applied to choose an algorithm given the type of rendering, primitives and effects desired. Hasenfratz *et al.* (2003) also mention that algorithm selection is based on a particular application's constraints. Comprehensive knowledge about algorithms is needed for developers to select the most ideal algorithm for a given situation. We realised that the collection and presentation of such knowledge is needed, but that it will not suffice. It often happens that parameters determining the ideal algorithm change from time to time within one application. To assist developers, an automated system to control the real-time selection of shadow rendering algorithms based on environmental conditions is needed. We aim to present such a solution.

1.3 Dissertation Structure

The current chapter presents the general research domain, research problem and overall dissertation structure with Chapter 2 presenting an analysis of various light sources (point lights, spotlights, ambient lights and parallel lights) followed by a discussion of different reflection models such as the empirical Phong model and the ambient, specular and diffuse reflection models. Following this discussion, the chapter investigates a number of shadow-generating algorithms, particularly focusing on the rendering of shadows by means of stencil shadow volumes and depth stencil testing. These elements are core to understanding subsequently discussed topics, especially those dealing with the performance improvement and quality of shadow rendering algorithms.

Chapter 3 extends this discussion to include implementation details for various shadow rendering algorithms, specifically the stencil shadow volume algorithm, the shadow mapping algorithm and a number of hybrid approaches such as McCool's shadow volume reconstruction using depth maps, Chan and Durand's hybrid algorithm for the efficient rendering of hard-edged shadows, Thakur et al's elimination of various shadow volume testing phases and Rautenbach et al's shadow volumes, hardware extensions and spatial subdivision approach.

Chapter 4 presents the critical analysis and detailed benchmarking of the previously discussed shadowing techniques. The chapter also looks at the performance of several optimised shadow volume and shadow mapping routines. The knowledge base of our expert system draws heavily on these experimental results.

Chapter 5 focuses on the theoretical aspects of expert systems, their architecture and a number of advantages and disadvantages inherent to their use. Following this discussion, it deals with the concept of fuzzy-logic and its use in the dynamic selection of shadow rendering algorithms.

Chapter 6 discusses the previously mentioned expert system implementation in much more detail. It also presents the critical analysis of our empirically derived system for the high-speed rendering of shadows. This analysis highlights not only the performance benefits inherent to the utilisation of this system, but also the practicality of such an implementation.

The final chapter features an overall summary of our work. It closes by discussing possible future work based on the presented research.

This discussion will thus analyse a vast number of shadow generation algorithms with the aim of highlighting the need for a system to control the real-time selection of shadow rendering algorithms based on environmental conditions. We present such a solution through the critical analysis of numerous real-time shadow rendering algorithms and the construction of an empirically derived system for the high-speed rendering of shadows. This critical analysis allows us to assess the relationship between shadow rendering quality and performance. It also allows for the isolation of key algorithmic weaknesses and possible bottleneck areas. Focusing on these bottleneck areas, we investigate several possibilities for improving the performance and quality of shadow rendering; both on a hardware and software level. Primary performance benefits are seen through effective culling, clipping, the use of hardware extensions and by managing the polygonal complexity and silhouette detection of shadow casting meshes. Additional performance gains are achieved by combining the depth-fail stencil shadow volume algorithm with dynamic spatial subdivision.

Using the gathered performance data, we are able to define a fuzzy logic-based expert system to control the real-time selection of shadow rendering algorithms based on environmental conditions. This system ensures the following: nearby shadows are always of high-quality, distant shadows are, under certain conditions, rendered at a lower quality and the frames per second rendering performance is always maximised.

Please note, unless otherwise stated, that all screenshots and/or illustrative images have been rendered using DirectX 10.0 based sample programs. The accompanying CD contains many of these sample applications, implementation source code and several videos (including a high-definition video showcasing the rendering framework).

Chapter 2

Basic Concepts: Shadows and Light

This chapter provides a brief overview of several foundational concepts needed for the ensuing discussion of shadow rendering algorithms – knowledge assumed in the remainder of this dissertation. It starts with an analysis of various light sources (point lights, spotlights, ambient lights and parallel lights) followed by a discussion of different reflection models such as the empirical Phong model and the ambient, specular and diffuse reflection models. Following this discussion, the chapter investigates a number of shadow-generating algorithms, particularly focusing on the rendering of shadows by means of stencil shadow volumes and depth stencil testing. Detailed coverage of these topics can be found in a large number of textbooks such as Rabin (2005) or Harbour (2004). Much of the material in this text has been taken from a recent textbook by the author (Rautenbach, 2008); Pharr and Fernando (2005) and Nguyen (2007) also serve as excellent texts for all the latest Graphics Processing Unit (GPU) programming techniques.

This chapter, as mentioned, focuses on a number of concepts inherent to the development of shadow rendering algorithms. We included this chapter with the reader's convenience and overall completeness in mind; individuals familiar with these concepts may proceed to subsequent chapters.

In this chapter we will investigate:

- Light sources
- Point lights
- Spotlights
- Ambient lights
- Parallel lights
- Emissive light
- Reflection models
- The ambient reflection model
- The specular reflection model
- The diffuse reflection model
- The Phong reflection model
- Shadow rendering algorithms
- Scan-Line polygon projection
- Blinn's shadow polygons
- Shadow mapping

- Shadow volumes
- Depth-pass testing
- Depth-fail testing
- Soft-edged shadows using penumbra wedges

2.1 Lighting

Before considering shadows, it's very important to briefly discuss the concepts of lighting and reflection (as there can be no shadows without light). The lack of lighting results in dull, flat looking object surfaces. Texture mapping helps to enhance the overall appearance of an object but fails to convey any real sense of depth. For example, when looking at the two flat objects in Figure 2.1 (a), it is clear that the three-dimensional nature of the scene, a wall positioned perpendicular on a floor, isn't being conveyed properly. Figure 2.1 (b) shows this same scene illuminated by a properly defined light source.

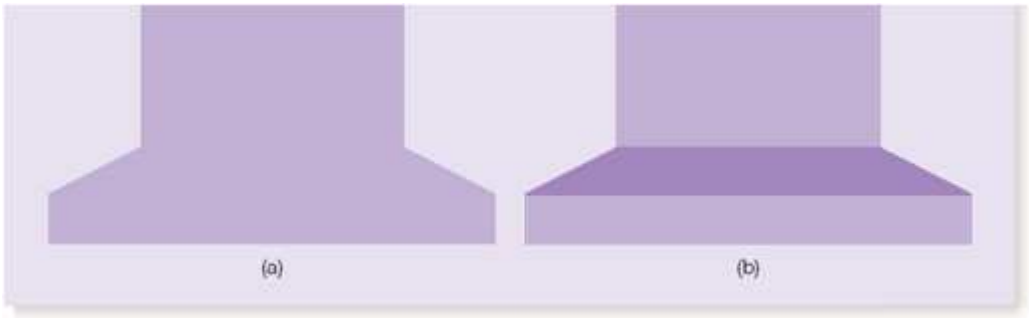


Figure 2.1 (a) Two rendered rectangles, the one representing a floor, the other a facing wall. (b) The same rectangles with lighting enabled.

This lack of depth is the result of uniform lighting, i.e. the equal illumination of all surfaces. Figure 2.2 (a) shows a uniform lit sphere and Figure 2.2 (b) the same sphere with basic lighting enabled. The shaded sphere is the result of graduations in the sphere's colour based on the colour of the light source. In this case the colour grey is incrementally decreased from dark grey to white.

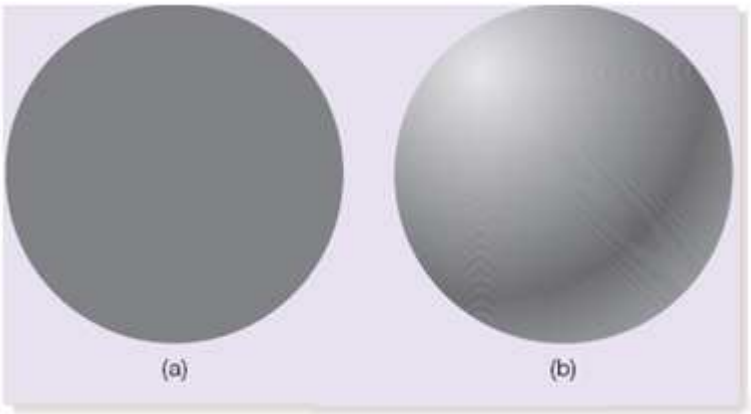


Figure 2.2 (a) A uniformly lit sphere and (b) a properly lit and shaded sphere.

Light can be emitted through either self-emission or reflection (Rautenbach, 2008). When looking at a light bulb it is obvious that we are predominantly dealing with self-emission. Light sources are categorised by their light emitting direction and the energy emitted at each wavelength – determining the colour of the light.

As also mentioned previously, objects can absorb or reflect light emitted from a light source depending on the reflecting object's material properties. Light will thus only be

"visible" when illuminated surfaces have the ability to reflect or absorb said light. Objects in computer generated graphical scenes are assigned so-called *Material properties.* These are user defined parameters built around rules determining the amount of scattering or reflection of incident light. Some surfaces, like a mirror, might reflect an incoming ray of light perfectly (hence appear shiny) while a carpet might reflect light in so many directions that it appears matte.

The type of light source also plays an important role in addition to the object's material properties. A *light type property* specifies the type of light to place in a scene. This property simply denotes a light source as a point light, spotlight or directional light (also called a parallel light). Lighting can thus be described as the interaction between a light source and an object's surface based on a pre-defined set of material properties. We will focus on each of these light source types in subsequent sub-sections.

A light source can be considered a geometric object, i.e. a simple light emitting surface. We can define a light emitting point on this surface (x, y, z) characterised by a wavelength energy value (λ) and an emitting direction (θ, ϕ) as shown in Figure 2.3.

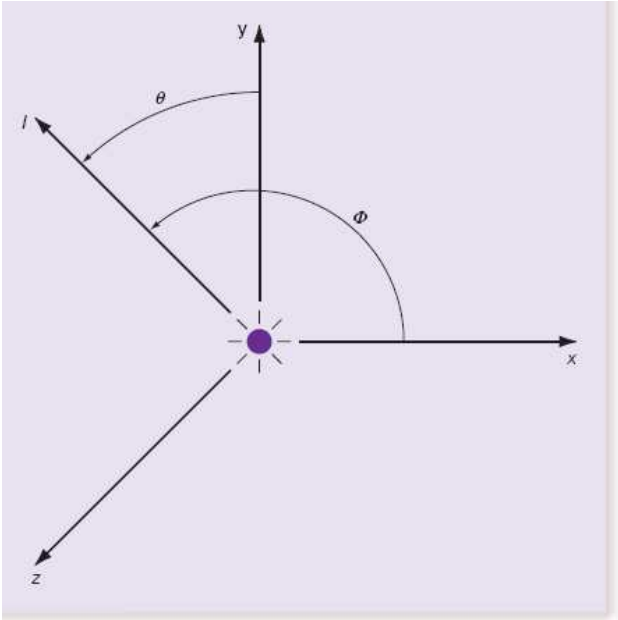


Figure 2.3 A basic light source characterised via six elements.

By combining these variables, we are able to define the *illumination function* $I(x, y, z, \theta, \phi, \lambda)$ used to describe any light source in terms of six variables. For example, say a surface is being illuminated by a light source; then we can calculate the overall illumination on this surface by integrating across the surface of the light source – thus incorporating the effect of the angle between the light source and reflection surface as well as the falloff distance (the distance from the light source to the reflecting surface). Figure 9.4 shows two distinct illumination functions for a pair of points located on the surface of a light source.

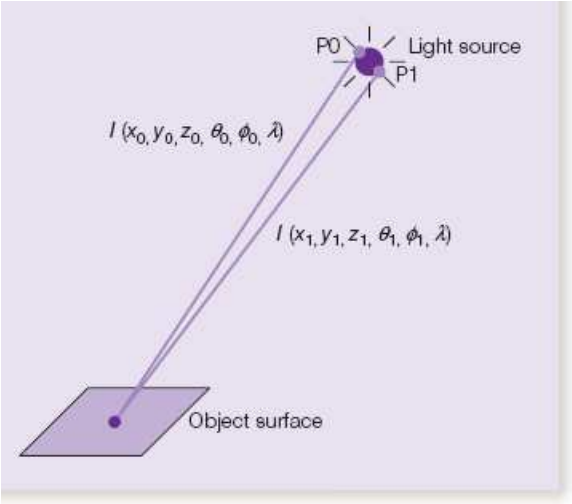


Figure 2.4 Two distinct illumination functions for a single light source.

Numerous colour intensities or shades can be described by additively combining various intensities of red, green and blue. Building on this, light sources can be defined using a similar red, green and blue colour component model. Each light source component is subsequently used to calculate the corresponding colour component of an illuminated surface. This three-component description is called *luminance* or *intensity*, and can be written using standard matrix notation with each component representing the intensity of either the red, green or blue colour component of the light source:

$$I = \begin{bmatrix} I_r \\ I_g \\ I_b \end{bmatrix}.$$

Furthermore, the overall lighting effect can be characterised by a lighting model (Whitted, 1980). A *lighting model* defines light-object interactions based on the type of light source and the material properties of the object. There are a number of commonly implemented lighting models and we will discuss several of these in section 2.2. For now, it's just important to note that the basic graphics pipeline is constrained to the use of just one lighting model, namely, the *fixed-function lighting* model. This lighting model is basically an extended version of the Phong lighting model. The dawn of shader programming allows for full programmability of the graphics pipeline, thus facilitating the implementation of custom user-specified lighting models such as Lambertian lighting, anisotropic lighting, Fresnel lighting and Blinn lighting.

2.1.1 Point Lights

A *point light* emits light uniformly in 360 degrees. Point lights have fixed colour and position values and are omnidirectional in nature. The objects illuminated by this light type appear either oversaturated (overly bright with a high contrast) or too dark – a

side effect easily corrected through the addition of ambient lights. The primary factor influencing brightness is the distance between the illuminated surface and the point light. Point lights are the easiest of all light types to implement, resulting in their widespread use regardless of their unrealistic simulation of real-life light sources. Figure 2.5 illustrates the effect of a point light illuminating a surface.

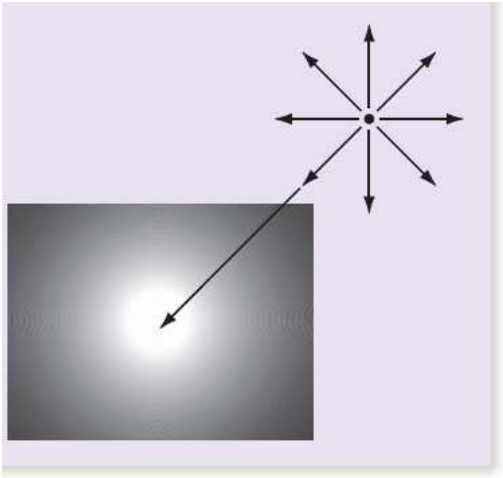


Figure 2.5 Point light illumination.

Using the previously discussed luminance function, we can define a point light located at point P_1 as follows:

$$I(P_1) = \begin{bmatrix} I_r(P_1) \\ I_g(P_1) \\ I_b(P_1) \end{bmatrix}.$$

Using this luminance function, we can calculate the level of illumination at a specific point, *k*, on a surface by multiplying the intensity of the light with the inverse square distance between the light source and illuminated surface:

$$I(k, P_1) = \begin{bmatrix} I_r(P_1) \\ I_g(P_1) \\ I_b(P_1) \end{bmatrix} * \frac{1}{|k - P_1|^2}.$$

2.1.2 Spotlights

Spotlights are specified by a colour, spatial position and some specific direction and range in which light is emitted. A spotlight is basically a point light with its emitting light constrained within an angle range. This range is defined using two cones: a bright inner cone and an encircling outer cone. The inner cone has a high intensity (correlating to the user-defined luminescence of the light source), with the outer cone used for fading or attenuating the light source's intensity in an outwards direction. This gradual reduction of light intensity is referred to as *falloff*. Falloff governs the

decrease in light intensity from the inner cone to the outer cone and a falloff value of 1.0 generally denotes an evenly distributed light intensity decrease. Figure 2.6 illustrates this diminishing property.

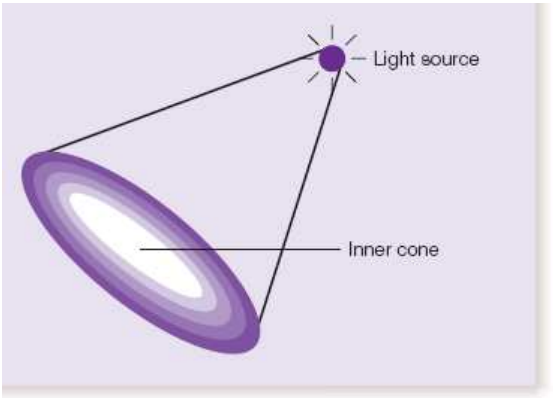


Figure 2.6 Spotlight falloff.

The intensity of a spotlight can be calculated by considering the angle between the direction of the light source and a vector to the point being illuminated. The simplest way of formulating this intensity is to calculate the cosine, to the power of *e*, of the direction angle:

$$I = \cos^{e}\theta, \quad \text{with } e \text{ representing exponential falloff.}$$

We can also calculate the dot product of the spotlight's direction vector and the vector to the point being illuminated. This calculation results in the cosine of the angle between these two vectors (shown in Figure 2.7):

$$\cos\theta = (spotlightDirectionVector) \bullet (vectorToSurface).$$

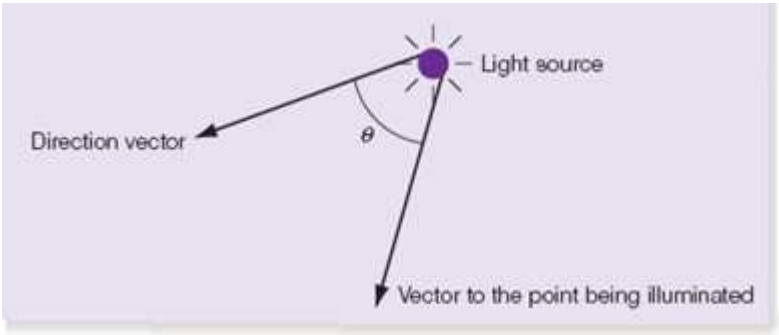


Figure 2.7 The relationship between the direction vector and the vector to the point being illuminated.

2.1.3 Ambient Lights

Ambient lighting provides a uniform level of illumination throughout a scene. Numerous large light sources are generally positioned in such a way as to scatter emitted light in all directions, thus making it impossible to determine the original position of the light source. Even though ambient light hitting a surface is scattered

equally in all directions, we can still determine the ambient intensity at each point on the surface.

This type of illumination has a luminance, *I*, which is the same for all points in the scene (with the manner of reflection being completely dependent on the material properties of a surface):

$$I_A = \begin{bmatrix} I_{Ar} \\ I_{Ag} \\ I_{Ab} \end{bmatrix}$$, with I_{Ar} the red, I_{Ag} the green and I_{Ab} the blue colour component of the ambient light intensity.

2.1.4 Parallel Lights

A *parallel* or *directional light* illuminates objects through a series of parallel light rays. These light sources can be considered as point lights located a significant distance from the surface of an object. Moving from one closely located object to another has little influence on the direction at which light hits the object. Sunlight can be considered a parallel light source due to it illuminating closely located objects at the same angle. Thus, the vector to the point being illuminated does not change a great deal when moving from one object to the next. We also use this direction vector to describe the light source. Figure 2.8 illustrates a parallel light source.

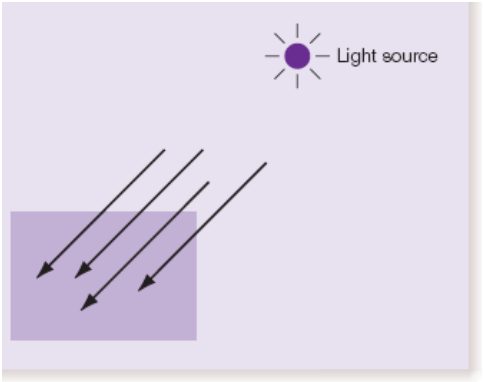


Figure 2.8 A parallel light.

Parallel lights do not exhibit attenuation or range properties. Consequently, they do not require any calculations dealing with illumination effects such as falloff. They are thus excellent light sources when computational overhead is being considered.

2.1.5 Emissive Light

Emissive light is radiated (can be considered self-reflecting) light originating from an object's surface. This type of light blends with our other light types, resulting in a surface smoothly coloured through the combination of all global light colour components. An object coloured using emissive light appears flat and unshaded; this is due to emissive reflection not considering vertex normals or "incoming" light

direction. We can describe emissive lighting using a three-component intensity function:

$$I_E = \begin{bmatrix} I_{Er} \\ I_{Eg} \\ I_{Eb} \end{bmatrix},$$ with I_{Er} the red, I_{Eg} the green and I_{Eb} the blue colour component of the emissive light intensity.

2.2 Reflection

A surface is only visible when it has the ability to reflect or absorb light. This ability is the result of the surface's material properties, i.e. rules determining the amount of scattering and/or reflection of incident light (Rautenbach, 2008). We can specify material properties for any surface, the most common types being the Phong reflection model, ambient reflection, diffuse reflection, specular reflection and transparency.

The basic lighting model can be considered as a high-level equation summing an ambient, diffuse and specular component to calculate the colour of an object's surface:

Surface colour = ambient lighting term + specular lighting term + diffuse lighting term.

This surface colour is actually equal to the overall amount of light present in a scene, commonly called global illumination and extended to include an emissive lighting term, resulting in the following lighting model equation used to simulate a wide range of lighting conditions:

Global illumination = ambient lighting term
+ specular lighting term
+ diffuse lighting term
+ emissive lighting term.

We will now look at each of these lighting/reflectance components as functions of material properties (e.g. surface reflectance, colour) and light source properties (e.g. light direction, colour, position, attenuation).

2.2.1 Ambient Reflection Model

Ambient reflection, also called continuous reflection, occurs whenever light emitted from a source is reflected so much that its origin is impossible to determine. Ambient light is omnidirectional in nature. Omnidirectional light is radiated uniformly in all directions, or more commonly, it is light scattered uniformly in all directions. This is also the reason for ambient reflection being described as continuous reflection – it being continuous in all directions, affecting the entire surface in an equal fashion.

Thus, some of the light hitting a surface is absorbed while the rest is reflected – resulting in ambient reflection. Also, every point in a scene receives the same amount of ambient lighting, with only the reflection of this light varying. Figure 2.9 illustrates this concept.

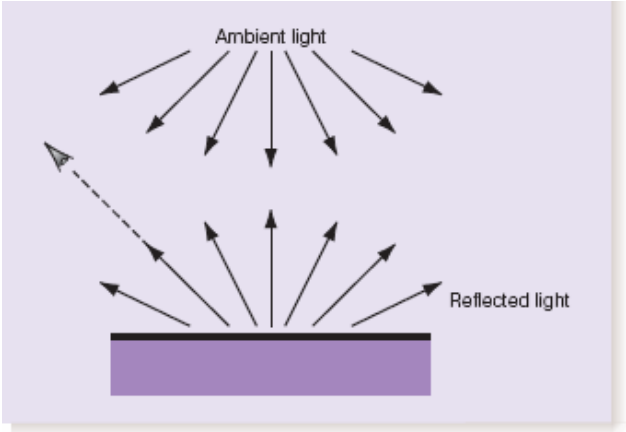


Figure 2.9 Ambient reflection

The problem with ambient reflection is that illuminated objects appear rather flat and unshaded; Figures 2.1 (a) and 2.2(a) show the classic appearance of ambient lit surfaces.

This 'flatness' is the result of ambient lighting not factoring in vertex normals or the direction, position, range, and additional light source properties such as attenuation or falloff. Ambient reflection is thus the most computationally efficient of all the reflection models. The ambient reflection coefficient is an indication of the reflected amount and is comprised out of red, blue, and green ambient reflection coefficients collectively. The equation for calculating ambient lighting factors in the material's ambient reflectance and the colour of the incoming ambient light:

Ambient lighting term = material's ambient reflectance x incoming ambient light colour.

We can also define the intensity of ambient reflection using the ambient luminance function (IA), the incoming ambient light colour (I) and the material's ambient reflectance consisting of three reflection coefficients – RAr, RAg and RAb, representing the red, green, and blue ambient reflection coefficients, respectively:

$$\begin{bmatrix} I_{Ar} \\ I_{Ag} \\ I_{Ab} \end{bmatrix} = \begin{bmatrix} R_{Ar} \\ R_{Ag} \\ R_{Ab} \end{bmatrix} * \begin{bmatrix} I_r \\ I_g \\ I_b \end{bmatrix}.$$

2.2.2 Specular Reflection Model

Specular reflection occurs whenever light, from a single incoming direction, is reflected at a single outgoing direction. Specular reflection is characterized by bright

highlights on the surface of an object reflected in the direction of the view vector. This concept is illustrated in Figure 2.10.

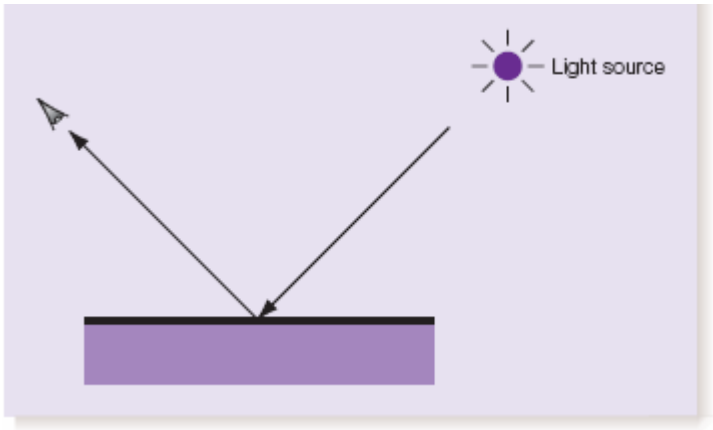


Figure 2.10 Specular reflection

Specularity can be defined the amount of shininess exhibited by an object with the level of specular reflection attributed to a user definable value, namely, the shininess coefficient. The bigger this coefficient, the smoother the object's surface and the closer we are to a perfect mirror. For example, values ranging from 100 to 500 represent most metallic surfaces while smaller values represent materials with broader highlights such as plastic and wood. Figure 2.11 shows several spheres with specular highlights.

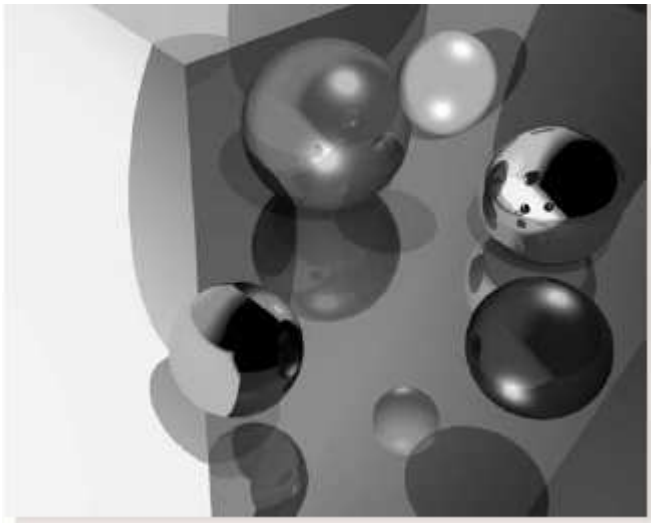


Figure 2.11 Examples of specular highlights

To calculate specular reflection we need information about both the incoming light direction and location of the viewer as well as the colour properties of the material, light source and shininess of the surface. The equation for calculating specularity is:

$$\begin{aligned}\text{Specular lighting term} = {} & \text{material's specular colour} \\ & \times \text{colour of incoming specular light} \\ & \times \text{geometryFacingFlag} \\ & \times (\max(\text{normalized surface normal} \\ & \quad \bullet \text{ normalized halfway vector}, 0))^{\text{shininess}}\end{aligned}$$

The *geometryFacingFlag* element is a flag ensuring that specular highlights are limited to geometry facing a light source – its value is calculated by taking the dot product between the normalized surface normal and the normalized vector pointing to the light source. If this dot product is greater than zero then the *geometryFacingFlag* element is set to 1, otherwise 0. The *normalized halfway vector* element is the vector halfway between the normalized vector pointing towards the viewpoint and the normalized vector pointing in the direction of the light source. Specular highlights are prominent when the angle between these two vectors is small. Figure 2.12 shows the vectors used in the calculation of this specular term.

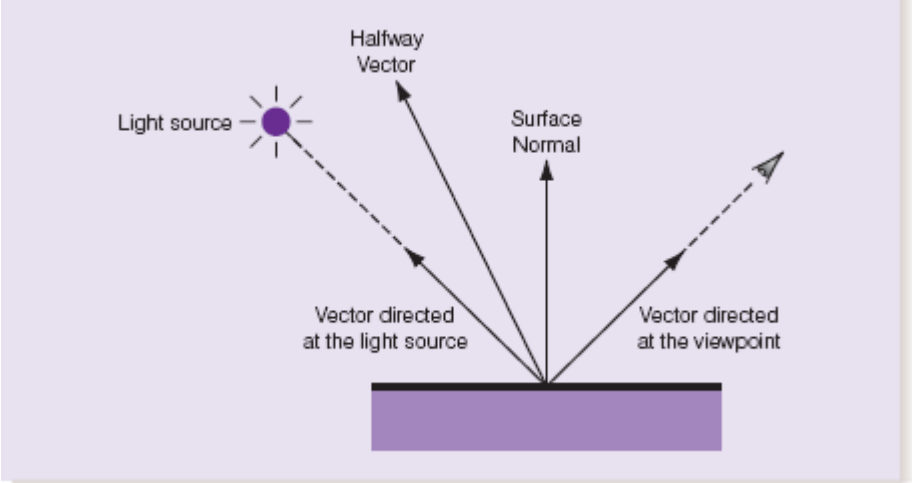


Figure 2.12 Vectors used in the calculation of the specular term

Alternatively we can define the intensity of specular reflection using the specular luminance function (I_S); the angle between the reflection vector (the direction of a perfectly reflected ray) and the vector directed at the viewpoint; the intensity of the specular light, I; the shininess coefficient, α; and Rs, the fraction of the incoming specular light being reflected:

$$\begin{bmatrix} I_{Sr} \\ I_{Sg} \\ I_{Sb} \end{bmatrix} = \begin{bmatrix} R_{Sr} \\ R_{Sg} \\ R_{Sb} \end{bmatrix} \begin{bmatrix} I_r \\ I_g \\ I_b \end{bmatrix} \cos^{\alpha}\phi.$$

2.2.3 Diffuse Reflection Model

Diffuse reflections occur when incoming light is reflected in arbitrary directions. The main contributing factor to this form of reflection is an uneven or rough surface. A diffuse surface appears identical to all viewers, regardless of their respective point of view. This type of reflection is common for matte or uneven surfaces (such as carpets or brushed metal) and is used for shading surfaces in such a way as to convey a sense of depth.

Diffuse reflection is a function of the incoming light direction and surface normal, in other words, the reflection of incoming light is dependent on the surface roughness and incoming light angle. The equation for calculating diffuse lighting is:

Diffuse lighting term = material's diffuse color
x color of incoming diffuse light
x max(normalized surface normal
• normalized vector towards light,0)

The dot product between the normalized surface normal and normalized vector pointing towards the light source gives the measure of incident light received by the surface – the smaller the angle between these two vectors, the greater the dot product result, and the greater the amount of incident light falling on the surface. The max (normalized surface normal. normalized vector towards light, 0) element in the equation ensures that only surfaces facing a light source reflect some diffuse lighting – surfaces facing away from a light source result in a negative dot product. Figure 2.13 shows a diffuse surface with the normalized surface normal and normalized vector pointing at the light source.

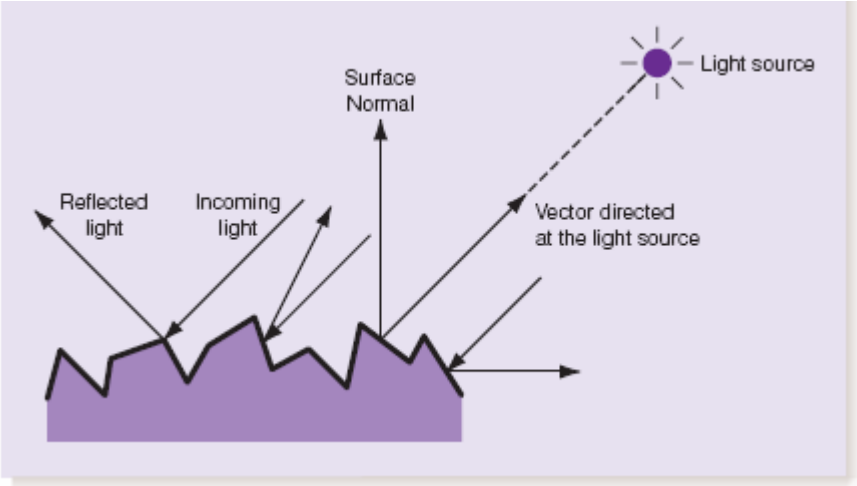


Figure 2.13 Diffuse reflections

We can also define perfect diffuse surfaces, i.e. surfaces reflecting light in no particular direction. These surfaces, also called Lambertian surfaces, are generally so rough that it is mathematically impossible to determine a preferred angle of reflection. Also, Lambertian light has a consistent intensity regardless of the distance between the reflecting surface and light source.

Perfect diffuse reflection can be modelled using Lambert's cosine law. This law states that the reflection or radiance observed from a perfect diffuse surface is directly relative to the cosine of the angle between the vector directed at the light source and the surface normal:

$R_D \propto \cos\phi$, where R_D is the diffuse reflection and ϕ is the angle between the vector directed at the light source and the surface normal.

Simply put, Lambert's law states that a perfectly diffuse surface always reflects the same amount of light, regardless of the viewing angle. For example, say a surface is being illuminated using a parallel light source, when this light is positioned perpendicular to the surface; the surface will appear brightly lit. Placing this light source at, say, a 135 degree angle will result in a more dimly lit surface due to the light rays covering a larger surface area. Figure 2.14 illustrates Lambert's cosine law.

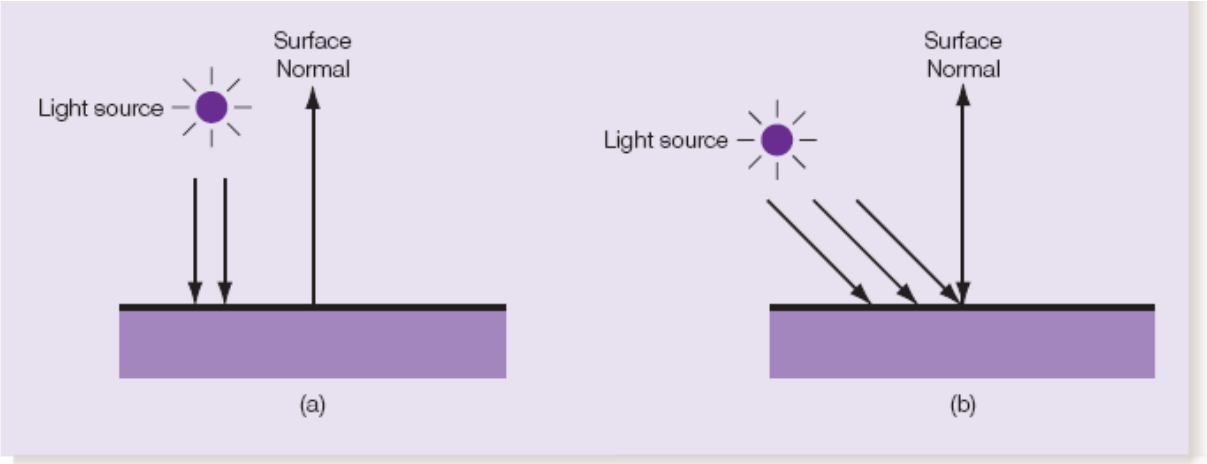


Figure 2.14 A perfect diffuse surface being illuminated by (a) a light source positioned perpendicular to the surface and (b) a light source positioned at a 135 degree angle

2.2.4 The Phong Reflection Model

The *Phong reflection model*, also loosely called Phong shading, was developed in 1973 by Bui Tuong Phong (the late computer graphics researcher and pioneer) and later extended to include a halfway vector in the calculation of the specular term by Jim Blinn. The Phong model is an illumination model that controls the shading of individual pixels; it is computationally efficient and leads to realistic looking reflections. Phong's goal was to create realistic looking objects in as close to real time as possible. The Phong reflection model basically combines ambient, specular and diffuse lighting components to closely approximate real world reflections. This concept is shown in Figure 2.15. We can consequently write the combination of these lighting terms as:

$$I = Ia + Id + Is.$$

Figure 2.15 Combining the lighting terms, producing a Phong reflection

Mathematically, the Phong reflection model considers reflected light as a function of the cosine between the surface normal and the incoming light direction. More precisely, the colour value of a point on the surface being illuminated is a function of four vectors, as shown in Figure 2.16: the normal vector at this point, the vector directed at the viewpoint, a vector directed at the light source, and the reflection vector (indicating the direction of a perfectly reflected ray).

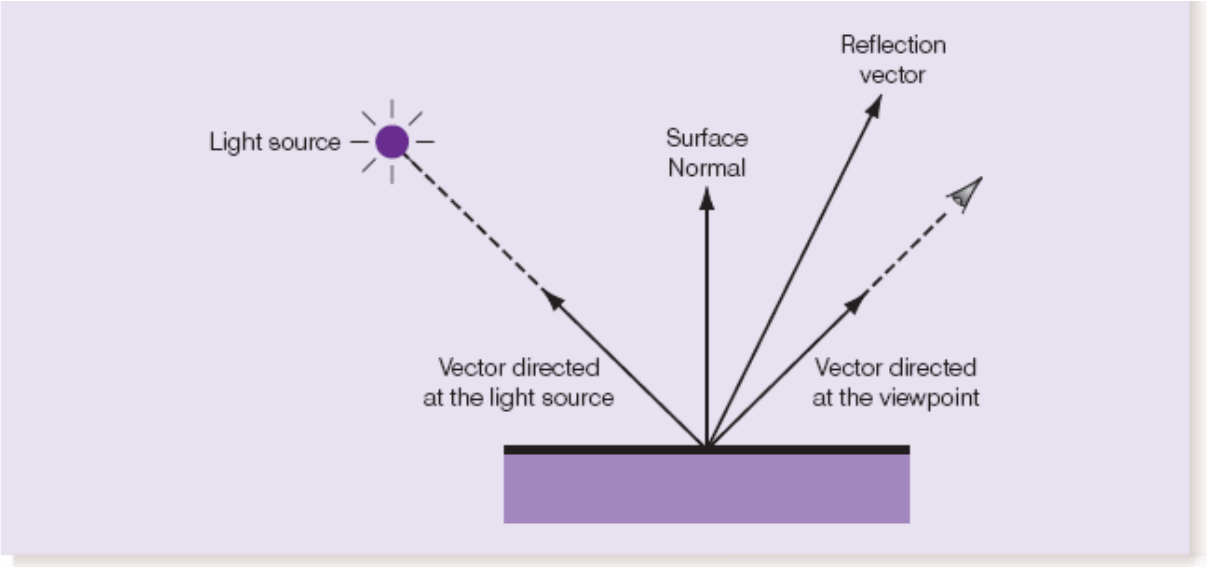


Figure 2.16 Vectors used in the calculation of the Phong reflection model

The following equation can be used to calculate the Phong reflection of a point on the surface of an object:

$$\textit{Phong reflection} = \mathrm{k_a} i_a + \sum (k_d (L \cdot N) i_d + k_s (R \cdot V)^{\alpha} i_s)$$

with k_a the material's ambient reflectance, i_a the colour of incoming ambient light, k_d the material's diffuse reflectance, L the vector directed at light source, N the surface normal, i_d the colour of incoming diffuse light, k_s the material's specular reflectance, R the reflection vector, V the vector directed at the viewpoint, a the shininess coefficient and is the colour of incoming specular light. The Phong reflection, using this equation, is typically calculated for individual intensities of red, green, and blue. The sum component in the above given equation defines a set of light sources. The effect of each light source, on the point being illuminated, is thus considered by the equation.

2.3 Introduction to Real-time Shadow Generation

Real-time shadow generation contributes heavily towards the realism and ambience of any scene being rendered. Research dealing with the calculation of shadows has been conducted since the late 1960s and has picked up great momentum with the evolution of high-end dedicated graphics hardware. Shadows are produced by opaque or semi-opaque objects obstructing light from reaching other objects or surfaces. A *shadow* is a two-dimensional projection of at least one object onto another object or surface (Crow, 1977). The size of a shadow is dependent on the angle between the light vector and light blocking object. The intensity of a shadow is in turn influenced by the opacity of the light-blocking object. An opaque object is completely impenetrable to light and will thus cast a darker shadow than a semi-opaque object. The number of light sources will also affect the number of shadows in a scene (with the darkness of a shadow intensifying where multiple shadows

overlap). Figure 2.17 illustrates shadow generation, specifically the implementation of stencil shadow volumes – a popular shadow rendering technique.

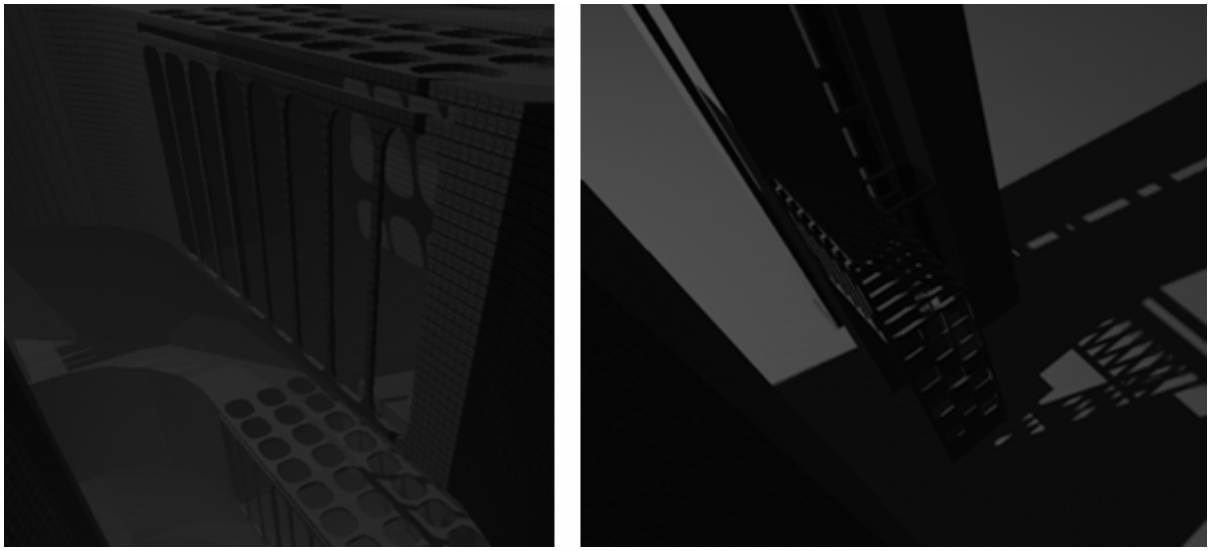


Figure 2.17 Example of stencil shadowing – note the darkening of overlapping shadows.

The drive towards realism has led to the development of many shadowing algorithms. Some of these algorithms, like shadow mapping and shadow volumes, are more successful than others. The success of an algorithm is dependent on the balance between speed and realism and techniques like shadow mapping and stencil shadow volumes are particularly amenable to hardware implementation – thus freeing the CPU of a substantial processing burden and making the real-time rendering of shadows feasible (Kilgard, 1999). Other shadowing approaches, such as the one proposed by Boulanger et al (2003), have in turn focussed on visually pleasing approximations for computationally expensive natural scenes.

Looking at shadows from a foundational perspective reveals them as a product of an environment's lighting. Shadows can have either hard or soft edges. This is dependent on the type of light source used and the distance between the light source and object. In the case of soft shadows we differentiate between both an umbra and penumbra. The darkest area of a shadow, receiving no light at all, is referred to as the *umbra* with the *penumbra*, receiving a small amount of light, indicating the partially shadowed edge (Akenine-Möller et al, 2002). Figure 2.18 illustrates a shadow's umbra and penumbra.

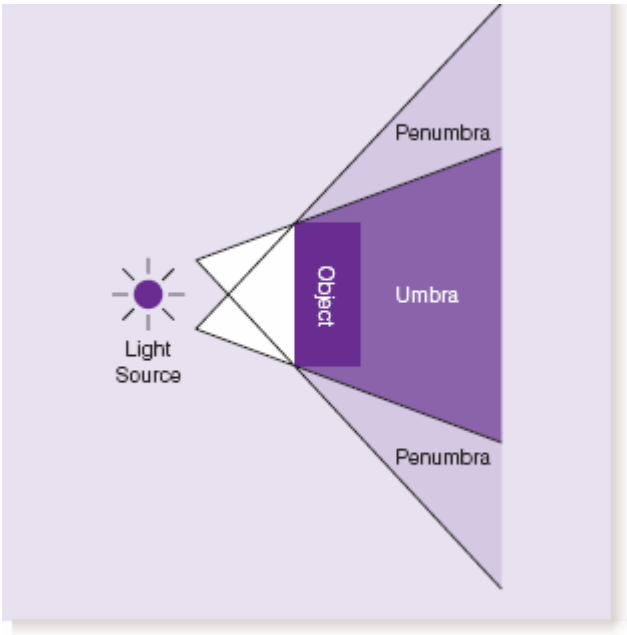


Figure 2.18 A soft shadow with related umbra and penumbra.

It should be noted that there is always a gradual intensity transformation from the umbra to penumbra (Akenine-Möller et al, 2002). However, the fading of the shadow (as its distance from the casting object increases) need not necessarily be gradual. Point lights will, for example, produce non-fading hard-edged shadows, with ambient light sources producing soft-edged shadows fading into the distance. The area of a light source also affects the gradual softening of shadows. The larger the light source's area, the more quickly the shadow grades off. Figure 2.19 shows the difference between shadows produced by point and ambient light sources.

Figure 2.19 (a) Hard-edged shadow produced by a point light source. (b) Soft-edged shadow produced by an ambient light source.

We will now investigate several shadowing algorithms, including the fundamentals of shadow volumes and shadow mapping. The first two algorithms, namely scan-line polygon projection and Blinn's shadow polygons, are historic in nature. We describe these algorithms here not only for the sake of completeness but also since some of the elements introduced by them form the basis of general shadow computation. These first two techniques aren't suited for real-time implementations. However, more recent algorithms such as stencil shadow volumes and hardware shadow

mapping remedy this situation by emphasising the balance between processor efficiency and realism.

It is necessary to note, before continuing, that shadowing remains one of the most processor intensive tasks and despite each technique's limitations, it is important to consider each algorithm with its intended application area in mind.

2.4 Shadow Rendering Algorithms

2.4.1 Scan-Line Polygon Projection

A quite complex, and now mostly redundant shadow algorithm was introduced by Appel (1968) and further developed by Bouknight and Kelley (1970). This algorithm, commonly known as *scan line polygon projection*, adds shadow generation to scan-line rendering. A *scan-line algorithm* operates on a row-by-row basis, as opposed to a pixel-by-pixel or polygon-by-polygon basis. A *scan-line* itself is a single line or row composed of a series of successive pixels stored in an array or list. The overall image is rendered as a result of the consecutive downwards repositioning of the scan-line. To enable both pre-rendered and real-time shadow generation via scan-line algorithms, it is necessary to append the original algorithm with a pre-processing stage. This pre-processing stage builds up a secondary data structure linking all the polygons that will cast a shadow on some other polygon.

The scan-line projection algorithm has an additional stage where all the polygons of a scene are projected onto a sphere centred at the light source (the centre of projection). This allows for the identification of all polygons casting shadows on other polygons. It is important to remember that, in a scene with k polygons, one will have at most $k(k - 1)$ shadows – the detection and elimination of polygon groups not interacting are thus of crucial importance. With all the shadow casting polygons linked in a secondary data structure, we can now project the edges of these polygons onto polygons intersecting the scan-line. A pixel's colour value is modified wherever the scan-line traverses one of these shadow edges. Hence, the light source (at the centre of projection) and shadow polygon cast a shadow onto the polygon intersected by the scan-line. The following cases denote whether a given pixel is in shadow or not:

1) The scan-line algorithm continues normally if no shadow casting polygon for the given pixel exists.
2) Decrease the brightness of the scan-line segment's pixels if a shadow casting polygon fully overlaps the intersected polygon.
3) If a shadow casting polygon partially overlaps the intersected polygon, subdivide the intersecting scan-line segment recursively until condition 1 or 2 is reached.

Scan-line polygon projection only allows for the generation of hard-edged shadows via point light sources. Figure 2.20 illustrates the above described process.

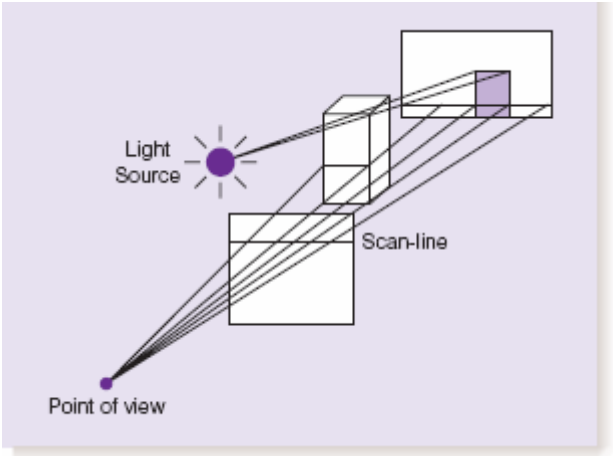


Figure 2.20 Scan-line polygon projection.

2.4.2 Blinn's Shadow Polygons

An extremely easy to use shadow generation technique was described by Blinn (1988). This method simply calculates the projection of an object on some base-plane. In short, a shadow cast by a point light and a polygon onto another polygon can be rendered by projecting the first polygon onto the plane of the second polygon (Blinn, 1988). The point light is in this case at the centre of projection and the resulting shadow is referred to as a *shadow polygon*. Figure 2.21 illustrates the projection of a shadow polygon (onto the *xy*-plane) with the light source located at the centre of projection.

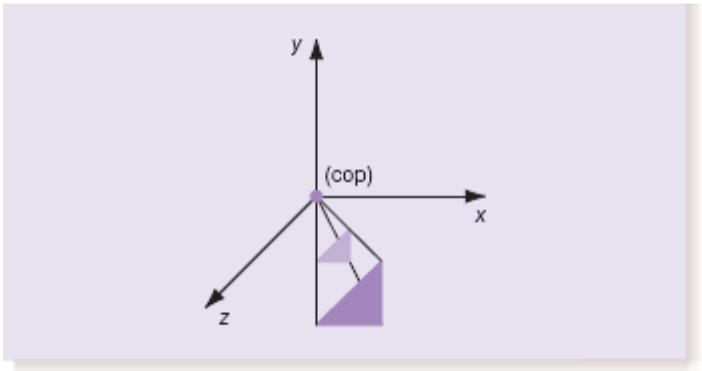


Figure 2.21 Shadow polygon with a point light source at the centre of projection.

The local illumination approximation states that if we have an infinitely positioned point light source, then we can consider its light rays as parallel (Phong, 1975). These rays, emanating from a light source located at the point (x_l, y_l, z_l), will cast a shadow at the point (x_s, y_s, z_s) based on the intersection of any point (x_o, y_o, z_o) located on an object positioned between the light source and some plane.

Generally though, if we have some finitely positioned point light, then we can translate the scene by some matrix, $T(-x_l, -y_l, -z_l)$, so that the light source is

positioned at the centre of projection. This translation yields the following projection matrix:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{-y_l} & 0 & 0 \end{bmatrix}.$$

After applying this projection matrix we have to translate the scene back to its original position with the generic translation $T(x_l, y_l, z_l)$. By concatenating the two translation matrices with the projection matrix, we are able to define the shadow projection (x_s, y_s, z_s) of the original point (x_o, y_o, z_o) as:

$$(x_l - \frac{x - x_l}{(y - y_l)/y_l}, 0, z_l - \frac{z - z_l}{(y - y_l)/y_l}).$$

The following steps outline the process of creating a shadow polygon:

1) Define and initialise the shadow projection matrix **M**.
2) Render the polygon normally.
3) Translate the light to the origin (centre of projection).
4) Calculate the projection of the object with the shadow projection matrix.
5) Translate everything back to their original positions.
6) Render the shadow polygon.

This method is often utilised to render the shadows of single polygons (Blinn, 1988). It is, however, only useful for the projection of shadows on flat surfaces, not for inter-object shadows. We will much rather implement an alternative method whenever objects are expected to cast shadows on other objects. For example, we could create a relatively uncomplicated shadow algorithm by simply modifying a hidden surface removal algorithm. The premise behind our modification would be that shadows are in fact areas hidden from light sources.

2.4.3 Shadow Mapping

Lance Williams introduced the concept of shadow mapping in 1978. His primary aim was the rendering of shadows on curved surfaces. *Shadow mapping* adds shadows to a scene by testing whether a particular pixel is hidden from a light source. It does this by first constructing a separate shadow Z-buffer for every light source and then storing the depth information of a scene in this buffer with the light source as view point. This depth information leads to a *depth image* or *shadow map* consisting of all the polygons not hidden from the light source. Hidden pixels are discovered through

a comparison with this depth image (Everitt et al, 2001). The shadow map partitions a light's view volume into shadowed and non-shadowed regions and we store this depth buffer image (shadow map) as a texture in the 3-D accelerator's texture unit. This texture is subsequently projected onto an area and/or object(s) for the shadow effect.

Although the shadow map is now stored in the display adaptor's texture memory, it must still be updated every time changes are made to the scene's light sources, geometry or object positions. However, no updating of the shadow map is required when altering the camera's point of view. We will typically partition the scene when implementing shadow maps, thus limiting the time it takes to update the depth image.

The final step of the algorithm is to render the scene via a Z-buffer algorithm. More specifically, if a pixel is not hidden from the light source then the related vertex is translated from the view point's screen space to light space (screen space with the light at the centre of projection). After all the vertices of an object have been translated, we have the object's spatial location from the light source's point of view.

The *x*- and *y*- coordinates of a translated vertex are used to index the shadow Z-buffer. Its *z*-component is used during the depth comparison test. This test simply compares a vertex's depth value to the corresponding value stored in the shadow map, determining whether the specific vertex will be shadowed or not. More explicitly, the vertex is in shadow if its depth value is greater than the value stored in the shadow map. For all other cases we can say that the vertex is closer to the light source than another arbitrary shadow casting surface and will thus be rendered without a shadow. Figure 2.22 shows a 3-D object and its resulting shadow map.

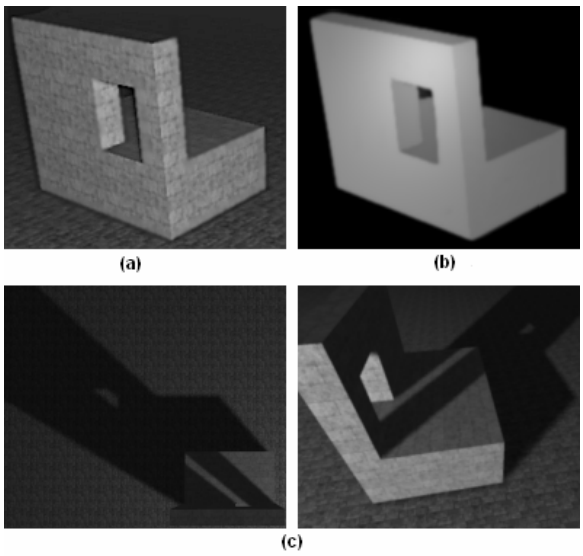


Figure 2.22 (a) Object as seen from the light's point of view (b) Object's depth map from the light's point of view (c) Shadow polygon rendered via the horizontal projection of the depth map.

Shadow mapping can be implemented as either a single- or multi-pass algorithm (Everitt et al, 2001). That is, if a fragment shader is used to render shadows by

performing the depth comparison test, then we will not require additional passes to produce the shadow maps. However, if we do not make use of programmable shaders (such as NVIDIA's Cg or DirectX's High Level Shader Language) then we won't have access to predefined lighting models (lit or shadowed) and will consequently have to implement an additional shadow map generation pass for each light source. In more complete terms, we can outline the dual-pass shadow mapping process as follows:

1) Create the shadow map by rendering the Z-buffer with regard to the light's point of view.
2) Draw the scene from the viewer's point of view.
3) For each rasterized fragment, calculate the fragment's coordinate position with regard to the light's point of view.
4) Use the x- and y- coordinates of step 3's translated vertex to index the shadow Z-buffer.
5) Do the depth comparison test, if the translated vertex's depth value (the z-value of step 3's translated vertex) is greater than the value stored in the shadow Z-buffer, then the fragment is shadowed, else it is lit.

Shadow mapping suffers from aliasing errors due to the use of a projection transformation mapping shadowed pixels to screen pixels, often causing changes in a pixel's screen size. This is a direct result of the Z-buffer algorithm's use of point sampling. The rendered shadow's edges are often jagged due to point sampling errors occurring during the calculation phase of the shadow Z-buffer. These errors are further amplified when accessing the shadow Z-buffer for the projection of pixels onto the shadow Z-buffer map. The only way of minimising the visibility of a shadow's jagged edges is to implement some form of pre-filtering and to use very large (high resolution) shadow maps.

2.4.4 Shadow Volumes

A *shadow volume* is a volumetric area defined by light rays extending outwards about the silhouette edge of an object (Crow, 1977). All the objects positioned within a shadow volume are hidden from the light source and are thus in either full or partial shadow. The contour of an object's surface is defined as a *silhouette edge* when the normal vector of the surface is perpendicular to the view vector (Everitt et al, 2002). A silhouette edge can more generally be considered as an outline or edge separating a front- and back-facing surface (Heidmann, 1991). The shape of the shadow volume is determined by the shape of the object's silhouette edge and a shadow volume is made up of so-called "invisible" shadow polygons. We refer to these shadow polygons as "invisible" since they are never rendered and only used to determine the shadowed areas. Shadow volumes are theoretically infinite volumes produced by polygons; however, for practical usability we intersect an infinite shadow volume with the view volume to produce a finite front- and back-capped shadow volume. Figure 2.23 shows the silhouette edge of a cube with Figure 2.24 illustrating the capping of a semi-infinite shadow volume.

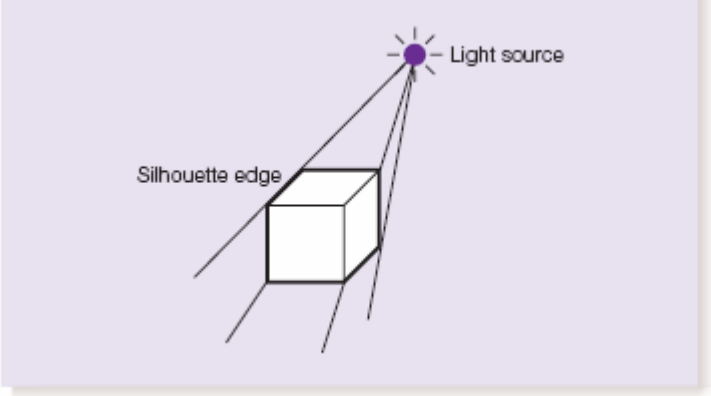


Figure 2.23 A simple silhouette edge.

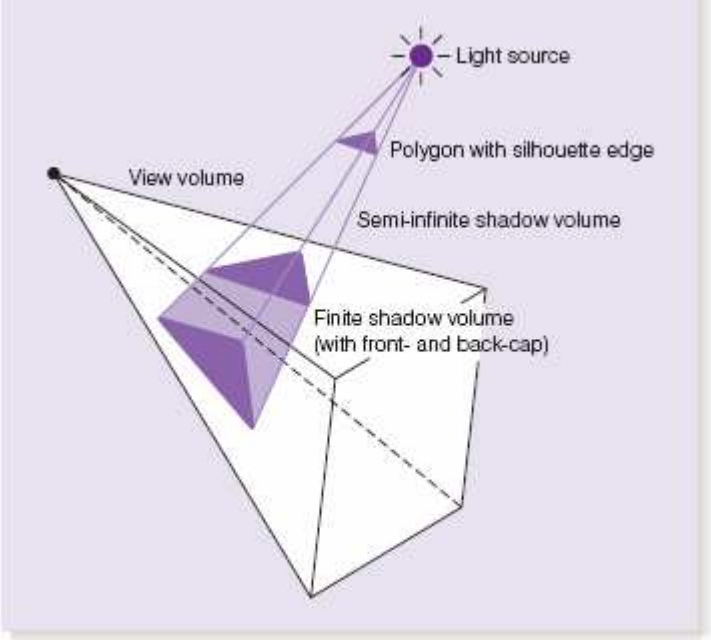


Figure 2.24 Construction of finite shadow volume.

The original shadow volume concept was introduced by Frank Crow in 1977. He defined a shadow volume as three-dimensional area occluding objects and surfaces from a light source. This original approach has since been extended to incorporate the generation of soft-edged shadows, including revision of the algorithm to utilise modern-day 3-D acceleration capabilities. The advent of dedicated 3-D acceleration hardware and the direct control of this hardware via APIs such as OpenGL and Direct3D have significantly contributed to the use of shadow volumes in modern computer games such as id Software's *Doom 3* and Bioware's *Neverwinter Nights* (Carmack, 2000).

The first feasible real-time shadow volume algorithm was introduced by Tim Heidmann in 1991. His algorithm made use of the 3-D accelerator's stencil buffer – effectively limiting the render area (called stencilling). The *stencil buffer* controls rendering by enabling or disabling drawing to a specific pixel. Heidmann discovered that the stencil buffer could be used to count the number of front- and back-facing shadows in front of an object if we rendered the shadow surfaces in two passes. By

counting these shadow surfaces we are able to determine whether an object's surface is in shadow or not. Heidmann's technique became known as the *depth-pass* stencil mask generation algorithm.

The general Heidmann stencil shadow volume process is summarised by the following phases:

1) Assume the scene in entirely shadowed.
2) Render the shadowed scene.
3) Calculate the shadowed scene's depth information.
4) Use this depth information to define a mask via the stencil buffer to indicate the lit areas.
5) Assume the scene is entirely lit.
6) Render the lit scene, applying the stencil buffer mask to cast the shadows.

There are two variations to the depth-pass technique, namely, depth-fail and exclusive-or (the latter of which is omitted due to its failure in dealing with intersecting shadow volumes). All shadow volume algorithms follow the above described shadow generation process and differ only in their approach of calculating the stencil mask. The depth-pass and depth-fail stencil shadow volume algorithms are described in detail below.

2.4.4.1 Depth-pass

Shadow volume algorithms operate on a per-pixel basis, performing a shadow test for every pixel in the frame buffer. We refer to all the data needed for the rendering of a pixel (stored in the frame buffer) as a fragment. Our algorithms will thus focus on all rasterized fragments to determine whether a specific fragment is in shadow or not. In more complete terms, we can write the above outlined stencil shadow volume process as follows:

1) For each rasterized fragment, render the fragment using ambient lighting, updating the Z-buffer after each fragment has been rendered.
2) Now we have to compute which fragments are in shadow. We once again look at each rasterized fragment, rendering the fragment as lit if not shadowed.

We can use the depth-pass method to test whether a fragment is in shadow or not. This method computes the fragments in shadow by generating a stencil mask. Using the stencil buffer, we count the number of front- and back-facing shadows in front of an object by rendering the front- and back-faces of the shadow surfaces in two passes. By counting these shadow surfaces we are able to determine whether an object's surface is in shadow or not. If there are more front-facing shadow surfaces than back-facing ones, then we can conclude that a shadow is projected onto an

object. The following process is used to compute the number of fragments in shadow:

1) For each rasterized fragment, render the fragment using ambient lighting, updating the Z-buffer after each fragment has been rendered.
2) Determine the silhouette edges of a shadow casting object. Following this the shadow volume polygons (shadow surfaces) are calculated (from the light source using the silhouette edges of the shadow casting object). These two steps are performed for each shadow casting object.
3) Now deal with the front- and back-facing shadow surfaces with regard to the point of view, **incrementing** the stencil buffer value for each front facing shadow surface if the depth-test **passes** (depth-pass using the Z-buffer) – counting the shadows in front of the object. Following the test for front-facing shadow surfaces, we focus on each back-facing shadow surface with regard to the view point – **decrementing** the stencil buffer value if the depth-test for a specific shadow surface **passes**.

Following the above process, we simply have to check the stencil buffer value for each fragment to identify the fragments in shadow. If a fragment's stencil buffer value is greater than zero then we need not draw this fragment during the second rendering pass – hence causing the fragment to be in shadow. Figure 2.25 illustrates the above described process:

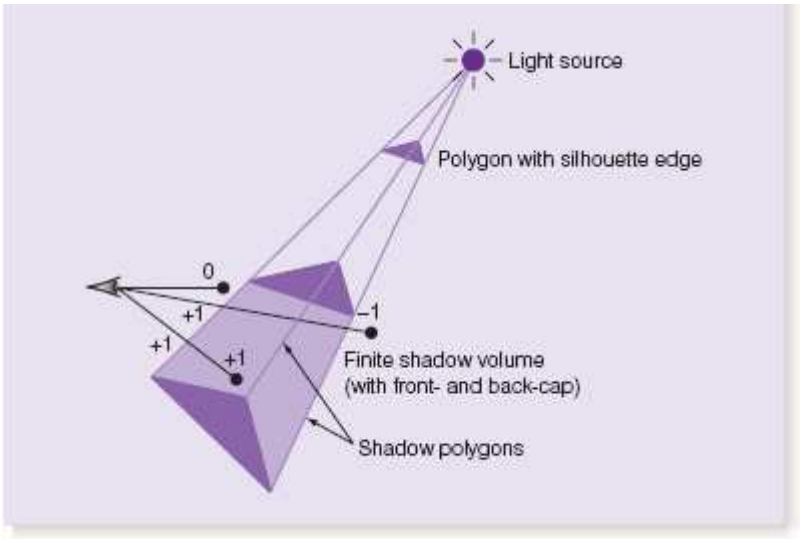


Figure 2.25 Testing whether a fragment is in shadow.

The described depth-pass process is extremely efficient; however, certain issues become apparent upon implementation. The most common problem occurs whenever the point of view (camera or viewer) is positioned within a shadow volume. This leads to visibility of the shadow's back-face. The depth-test will pass in this case, causing the stencil buffer value to be decremented, thus becoming -1 due to a back-face being visible prior to any front-facing shadow surfaces. This problem is referred to as *stencil counting inversion* and it can be resolved by capping the front of the shadow volume. Alternatively we can initialise the stencil buffer to 2^{K-1}, with K the

precision of the stencil buffer. These approaches are, however, less than efficient and the depth-fail technique is generally implemented as an alternative.

2.4.4.2 Depth-fail

The depth-pass approach computes the stencil buffer values by incrementing for front- and decrementing for back-facing shadow surfaces. The depth-fail approach modifies this calculation process (originally counting from the point of view) by counting from infinity. So, by reversing the depth and counting the shadow surfaces behind an object instead of those in front of it, we no longer face the *stencil counting inversion* issue. The only general issue with this approach is that we must cap the end of the shadow volume to avoid the condition where shadows point to infinity. The following process is used to compute the number of fragments in shadow:

1) For each rasterized fragment, render the fragments using ambient lighting, updating the Z-buffer after each fragment has been rendered.
2) Determine the silhouette edges of a shadow casting object. Following this the shadow volume polygons (shadow surfaces) are calculated (from the light source using the silhouette edges of the shadow casting object). These two steps are performed for each shadow casting object.
3) Now deal with the front- and back-facing shadow surfaces with regard to the point of view, **decrementing** the stencil buffer value for each front facing shadow surface if the depth-test **fails** (depth-fail using the Z-buffer). Following the test for front-facing shadow surfaces, we focus on each back-facing shadow surface with regard to the view point – **incrementing** the stencil buffer value if the depth-test for a specific shadow surface **fails**.

Although the depth-fail method effectively avoids the stencil counting inversion issue it still requires the additional back-capping of shadow volumes. This results in some extra rasterization time which can lead to considerable performance slowdowns under certain conditions. It is thus in some cases more advantageous to use the depth-pass method while explicitly dealing with the cases where the point of view is located within a shadow volume. It is also often possible to increase the performance of a stencil shadow volume implementation by utilising some hardware extension such as NVIDIA's *depth bounds test* enabling the culling of shadow volume sections not affecting the visible area.

It is interesting to mention though that Kolic el al (2004) developed a shadowing technique purely focussing on the utilisation of current GPU advances. Their algorithm specifically deals with the casting of shadows on concave complex objects such as trees. Koloc et al (2004) formally state that "for those objects, silhouette calculation that is usually preformed by other shadow volume algorithms is complicated and poorly justified. Instead of calculations, it is better to assume a worst case scenario and use all of the edges for construction of the shadow volume mesh, skipping silhouette determination entirely. The achieved benefit is that all procedures,

i.e. the object and shadow calculation and rendering, could be done on GPU. The proposed solution for shadow casting allows open edges. Indexed vertex blending is used for shadow projections, and the only calculation required is determining projection matrices. Once created, shadow volume is treated like any other mesh." When Crow implemented and defined the original shadow volume model back in 1977, he simply did not have access to any of these modern hardware acceleration aids and hence did not develop the now commonly used stencil shadow volume algorithm with modern day graphics accelerators in mind.

Thakur et al (2003) also developed a discrete algorithm for improving the Heidmann original. Chapter 3 deals with this algorithm in detail. Another significant algorithmic improvement over the Heidmann original was made by Chan and Durand (2004). They specifically combined the strengths of shadow maps and shadow volumes to produce a hybrid algorithm for the efficient rendering of pixel-accurate hard-edged shadows. Their method uses a shadow map to identify pixels located near shadow discontinuities, using the stencil shadow volume algorithm only at these pixels. This approach is also dealt with in Chapter 3.

2.4.4.3 Soft-edged Shadows using Penumbra Wedges

Implementation of the above discussed shadow volume techniques always result in pixel-accurate hard-edged shadows. Soft-edged shadows can be simulated through the construction of several shadow volumes by translating the original light source to various positions close to that of the original. Following this we simply have to combine the resulting shadows. The problem with this approach is rendering performance due to shadow volume construction taking up a substantial amount of processor time. One solution is the calculation of *penumbra wedges* as proposed by Akenine-Möller and Assarsson (2002). A penumbra wedge is defined in place of a shadow polygon for each silhouette edge of an object – combining a series of these penumbra wedges result in the creation of a soft-edged shadow.

The penumbra wedge algorithm calculates the amount of light that reaches a certain point *p*. This amount of light intensity ranges from '0' to '1'. When the light intensity is '0' we can define the point *p* as fully shadowed or conversely as fully lit with a light intensity of '1'. For all other values we can define point *p* located within the penumbra region. The light intensity inside the penumbra region is calculated using a signed 16-bit buffer. This light intensity buffer is simply a high precision stencil buffer. The lower the number of bits used for the buffer, the higher the implementation's performance and the lower the number of shades in the penumbra region. The varying shade levels in the penumbra region are created by multiplying each light intensity value stored in this buffer with some value *s*. This value is normally chosen as '255' since colour buffers allow for 8-bits per component, leading to at least '256' on-screen penumbra wedges. The following process is used for calculation of the penumbra wedges (illustrated in Figure 2.28):

1) Initialise the light intensity buffer to '255' – indicating that the viewer is now positioned outside of the shadow volume.
2) Draw the scene using both specular and diffuse lighting.
3) Draw the penumbra wedges using the following algorithm:
    a. For some light ray, compute the entry and exit points on the outside penumbra wedge. This must be done for each visible fragment. The entry point is defined by an *x*- and *y*-coordinate, with the corresponding *z*-value stored in the Z-buffer.
    b. Transform this point to world space coordinates (the point's independent local coordinate system has now been transformed into a global coordinate system. This provides all the points with a shared global coordinate space – i.e. one point's position can be described in terms of another's and all user defined points can now be positioned within the same scene).
    c. Test whether the point is located within the penumbra region.
        i. If the point is located within the penumbra region, compute the light intensity of this point and the entry point, scaling the light intensity by subtracting the computed light intensity of the point located within the wedge from the entry point and multiplying this result by '255'.
        ii. Add the above calculated light intensity to the light intensity buffer.
4) Add ambient lighting to the rendered scene.

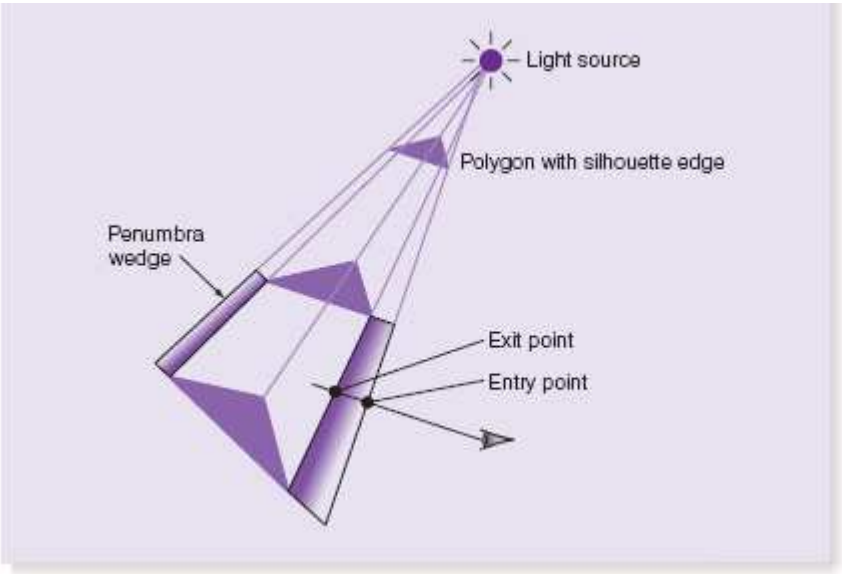


Figure 2.28 Locating a point within the penumbra region.

The possibility of overlapping penumbra wedges exists in situations where the volume is entered more than once. Such cases result in negative light intensity values, thus requiring the clamping of the values stored in the light intensity buffer to the range [0, 255]. It is also possible to leave the volume more than once whenever the viewer is located within the volume. By setting the maximum possible light intensity value to '255', we effectively avoid higher light intensities than that of the areas outside the volume – which clearly isn't possible.

Akenine-Möller and Assarsson's penumbra wedges algorithm (Akenine-Möller and Assarsson, 2002) can be implemented using either OpenGL or Direct3D. The main problem is the large vertex and pixel shader programs required, making true real-time performance only achievable on extremely high-end hardware. The following steps outline a hardware-accelerated implementation of the penumbra wedge algorithm:

1) Render the scene using either OpenGL or Direct3D.
2) Implement the wedge rasterization, initialising the Z-buffer prior to rasterization.
3) Rasterize the front facing triangles of the penumbra wedges – the entry point's plane is now identified.
4) Identify the exit point by calculating the ray's intersection with the back facing planes and picking the one closest to the ray.
5) Specify the point in world space coordinates via a transformation based on the Z-value.
6) Determine whether this currently selected point falls within a penumbra wedge or not by substituting the point's coordinates into the plane equations:
    a. If the point falls within a wedge, calculate the intersection distances from the point to the planes.

Brotman and Badler (1984) developed a similar algorithm for the generation of soft-edged shadows (adding penumbras to hard-edged shadows). They proposed the use of an enhanced Z-buffer algorithm, thus retaining the benefits inherent to the Z-buffer rendering approach. They extended the Z-buffer to represent a pixel location as a record of five fields. During the shadow polygon rendering phase, these pixel records are modified based on whether a point is lit or not. The penumbras are created by representing a distributed light source as a series of point light sources. This approach is processor intensive due to the combination of shadow volume calculations with Z-buffer memory access costs. Crowe's ideas were also extended by Bergeron (1985) to include non-planar polygons and objects.

2.5 Summary

In this chapter we focussed on the fundamentals of lighting and real-time shadow generation in detail. Light sources were introduced at hand of the role they play in the creation of properly lit and shaded objects. Building on this we presented the illumination function as a way to describe any light source in terms of six variables. We also discussed how a lighting model can be used to define light-object interactions based on the type of light source and the material properties of the object.

We subsequently investigated a number of light source types, specifically looking at point lights as light sources emitting light uniformly in 360 degrees, spotlights as point

lights emitting light within an angle range, ambient lighting as a way to provide a uniform level of illumination throughout a scene, parallel lights as sources illuminating objects through a number of parallel light rays and emissive light as self-reflecting light originating from an object's surface.

Following this we looked at several reflection models, namely, the ambient reflection model, the specular reflection model, diffuse reflection and the Phong reflection model. We specifically investigated these reflection models as functions of material properties (e.g. surface reflectance, colour, etc) and light source properties (e.g. light direction, colour, position, attenuation, etc).

Next we investigated real-time shadow generation and its contribution to the realism and ambience of rendered environments. We started by defining a shadow as the two-dimensional projection of at least one object onto another object or surface, subsequently looking at the physical properties of shadows and the technique of shadow casting in general.

Following this introduction we investigated a number of shadowing algorithms, specifically scan-line polygon projection, Blinn's shadow polygons, shadow mapping and stencil shadow volumes. Our discussion of Blinn's shadow polygons revealed it as an extremely easy to use shadow generation technique often utilised to render the shadows of single polygons on flat surfaces. We also considered the quite complex, and now mostly redundant scan-line polygon projection algorithm historically used for the generation of hard-edged shadows.

The chapter concluded by presenting the fundamentals of shadow mapping and stencil shadow volumes. Our shadow mapping discussion highlighted the general dual-pass shadow mapping technique. The shadow volume section presented the theory behind the construction of finite shadow volumes as well as the stencilling process, differentiating between the depth-pass and depth-fail methods used to test whether a fragment is in shadow or not. We also briefly touched on the generation of soft-edged shadows using penumbra wedges.

The discussion of these topics, as mentioned, was included to familiarise the reader with the basic concepts of real time scene rendering. The realism of rendered scenes is, of course, enhanced through the addition of shadows. The next chapter presents the implementation of various shadow rendering algorithms, specifically the stencil shadow volume algorithm, the shadow mapping algorithm and a number of hybrid approaches.

Chapter 3

Implementing Shadow Algorithms

Chapter 2 presented the theory behind numerous real-time shadow rendering algorithms and techniques with its particular focus being on the rendering of shadows by means of stencil shadow volumes and depth stencil testing. Chapter 3 extends this discussion to include implementation details for various shadow rendering algorithms, specifically the stencil shadow volume algorithm, the shadow mapping algorithm and a number of hybrid approaches such as McCool's shadow volume reconstruction using depth maps, Chan and Durand's hybrid algorithm for the efficient rendering of hard-edged shadows, Thakuret al's elimination of various shadow volume testing phases and Rautenbach et al's shadow volumes, hardware extensions and spatial subdivision approach as well as other documented enhancements. It specifically focuses on implementation details such as shadow volume and shadow map construction, the counting of front- and back-facing surfaces and the creation of silhouette and cap triangles, etc. All shadow generation techniques are implemented in C++ using Direct3D 10.0 and Microsoft's High Level Shading Language 4.0. The chapter illuminates selected portions of the code with the aim of providing the reader with a feel for the kind of coding needed and to illustrate the implementation details of these algorithms more clearly.

We discuss the implementation of these algorithms with the aim of incorporating them into a fuzzy-based expert system framework for the high speed rendering of shadows (discussed in Chapter 5).

In this chapter we will investigate:

- The Stencil Shadow Volume Algorithm
- The Stencil Buffer
- Enabling Depth-Stencil Testing
- Implementing Stencil Shadow Volumes
- The Shadow Mapping Algorithm
- Implementing Shadow Maps
- Shadow Volume Reconstruction from Depth Maps
- A Hybrid Algorithm for the Efficient Rendering of Hard-edged Shadows
- Elimination of various Shadow Volume Testing Phases
- Shadow Volumes and Spatial Subdivision

3.1 The Stencil Shadow Volume Algorithm

The stencil shadow volume benchmarking program consists of a relatively simple static cubic environment, a movable/interchangeable three-dimensional mesh object and a variable number of light sources (Figure 3.1, Figure 3.2).

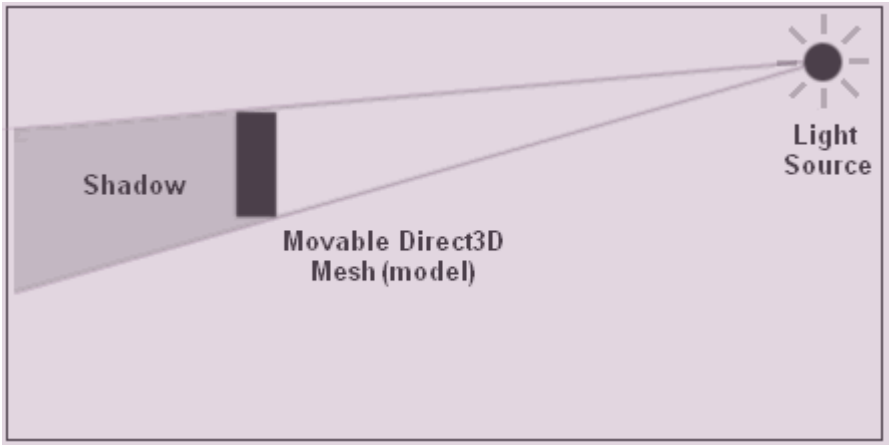


Figure 3.1 Top-down representation of the shadow volume evaluation environment.

Figure 3.2 Rendering a shadow by means of stencil shadow volumes (using one light source and three-dimensional mesh) – accurately cropped and skewed to fit the surrounding area.

The depth-fail stencil shadow volume algorithm is implemented using the following method (described in Chapter 2):

1) For each rasterised fragment, render the fragments using ambient lighting, updating the Z-buffer after each fragment has been rendered.
2) Determine the silhouette edges of a shadow casting object. Following this the shadow volume polygons (shadow surfaces) are calculated (from the

light source using the silhouette edges of the shadow casting object). These two steps are performed for each and every shadow casting object.

3) Now deal with the front- and back-facing shadow surfaces with regard to the point of view, **decrementing** the stencil buffer value for each and every front facing shadow surface if the depth-test **fails** (depth fail using the Z-buffer). Following the test for front-facing shadow surfaces, we focus on each back-facing shadow surface with regard to the view point – **incrementing** the stencil buffer value if the depth-test for a specific shadow surface **fails**.

We will now look at the stencil buffer and the depth-stencil testing process; two concepts crucial for the implementation of stencil shadow volumes. Building on this, we examine the implementation of stencil shadow volumes.

3.1.1 The Stencil Buffer

The stencil buffer is a buffer located on the 3-D accelerator that controls the rendering of selected pixels. *Stencilling* is the associated per-pixel test controlling the stencil value of each pixel via the addition of several bit-planes (one byte per pixel). These bit-planes, in association with depth-planes and colour-planes, allow for the storage of extra data – specifically the pixel's stencil value in the case of the stencil buffer. Stencilling is thus the process of selecting certain pixels during one rendering pass and subsequently manipulating them during another.

Stencilling can thus be described as the processes of defining a mask via the stencil buffer to indicate shadowed and lit pixel areas. With this information we apply the stencil buffer mask to update all the lit pixels, thus rendering shadows in the process. The stencil buffer allows for the manipulation of individual pixels, a property commonly used to create extremely accurate shadows. Use of the stencil buffer is, however, not limited to only the generation of shadows; it is also extensively used for reflections and has been widely supported since NVIDIA's RIVA TNT and the ATI RAGE 128 (circa 1998).

It is important to note the close relation between the stencil buffer and depth buffer. These two buffers are firstly located in physical proximity to each other (both commonly share the same physical area in the graphics hardware's memory). Secondly, the depth buffer is required to control whether a certain pixel's stencil value is increased or decreased based on the result of a depth test (pass/fail). The stencil buffer stores a stencil value for each pixel, similarly to the depth buffer storing the depth value of every pixel – both the stencil buffer and depth buffer values are required for rejecting or accepting rasterized fragments.

Enabling Depth-Stencil Testing

Before initialising the stencil buffer it is important to set the format of the depth stencil to **`DXGI_FORMAT_D24_UNORM_S8_UINT`** (previously **`D3DFMT_S8D24`** in DirectX 9). This DirectX Graphics Infrastructure (DXGI) component is responsible for defining the memory layout of each pixel making up an image. **`DXGI_FORMAT_D24_UNORM_S8_UINT`** is simply a DXGI enumeration type required by the **`DXUTDeviceSettings`** DXUT structure. DXUT simplifies the creation of a Direct3D device, the specification of windows and the handling of Windows messages. We set the **`AutoDepthStencilFormat`** member of the **`DXUTDeviceSettings`** structure as follows:

```
DXUTDeviceSettings* pDXUTDeviceSettings;

pDXUTDeviceSettings->d3d10.AutoDepthStencilFormat =
                                    DXGI_FORMAT_D24_UNORM_S8_UINT;
```

It is customary to clear the stencil buffer at the start of the rendering process (to erase previous changes). This is accomplished via the **`ClearDepthStencilView`** **`ID3D10Device`** (**`pID3D10Device`**) interface. **`ClearDepthStencilView`** clears the depth stencil using four parameters. Its first parameter is a pointer to the depth stencil we wish to clear, the second is a clear flag indicating the parts of the buffer to clear (**`D3D10_CLEAR_STENCIL`** for the stencil buffer and **`D3D10_CLEAR_DEPTH`** for the depth buffer), the third is the value we are clearing the depth buffer with (any value between '0' and '1') with the fourth parameter the value to clear the stencil buffer with. To initialise the first parameter (the depth stencil to be cleared), we simply call the **`DXUTGetD3D10DepthStencilView`** interface, resulting in a pointer to the **`ID3D10DepthStencilView`** interface for the current Direct3D 10 device:

```
ID3D10DepthStencilView* pDepthStencilView =
                                   DXUTGetD3D10DepthStencilView();

pID3D10Device->ClearDepthStencilView(pDepthStencilView,
                                     D3D10_CLEAR_STENCIL,
                                     1.0, 0);
```

In addition to clearing the stencil buffer, we also have to clear the depth buffer. The exact same process is used with **`ClearDepthStencilView`**'s second parameter being set to **`D3D10_CLEAR_DEPTH`**:

```
pID3D10Device->ClearDepthStencilView(pDepthStencilView,
                                     D3D10_CLEAR_DEPTH,
                                     1.0, 0);
```

The depth test's result is also needed in addition to that of the stencil test. As previously mentioned, the depth test result is required for controlling whether a certain pixel's stencil value is increased or decreased. If the depth test passes then

the tested pixel's depth value is overwritten by that of the incoming fragment. Both the depth test and stencil test results are combined for certain effects. The stencil test can simply fail, requiring no additional information, however, when the stencil test passes then the depth test can either fail or pass.

We can enable or disable both depth testing and stencil testing via the first (**DepthEnable**) and fourth (**StencilEnable**) parameters of Direct3D 10's **D3D10_DEPTH_STENCIL_DESC** structure. Furthermore, this structure allows us to specify the depth write mask (which controls the area of the depth-stencil buffer) that can be modified by depth data (**DepthWriteMask**), the depth function for comparing depth data against current depth data (**DepthFunc**), the stencil read mask specifying the area of the depth-stencil buffer for the reading of stencil data (**StencilReadMask**), the stencil write mask identifying the writeable depth-stencil buffer area (**StencilWriteMask**) and the stencil operations for both front-facing (**FrontFace**) and back-facing pixels (**BackFace**). These stencil testing operations (defined using the **D3D10_DEPTH_STENCILOP_DESC** structure) include the state when stencil testing fails, stencil testing passes and depth testing fails or when both stencil testing and depth testing passes.

The **D3D10_DEPTH_STENCIL_DESC** and **D3D10_DEPTH_STENCILOP_DESC** structures are defined as follows in the d3d10.h header file:

```
typedef struct D3D10_DEPTH_STENCIL_DESC {
    BOOL DepthEnable;
    D3D10_DEPTH_WRITE_MASK DepthWriteMask;
    D3D10_COMPARISON_FUNC DepthFunc;
    BOOL StencilEnable;
    UINT8 StencilReadMask;
    UINT8 StencilWriteMask;
    D3D10_DEPTH_STENCILOP_DESC FrontFace;
    D3D10_DEPTH_STENCILOP_DESC BackFace;
} D3D10_DEPTH_STENCIL_DESC;

typedef struct D3D10_DEPTH_STENCILOP_DESC {
    D3D10_STENCIL_OP StencilFailOp;
    D3D10_STENCIL_OP StencilDepthFailOp;
    D3D10_STENCIL_OP StencilPassOp;
    D3D10_COMPARISON_FUNC StencilFunc;
} D3D10_DEPTH_STENCILOP_DESC;
```

The default values, including the alternatives, for the members of the **D3D10_DEPTH_STENCIL_DESC** structure are given in the table below (Microsoft, 2008):

Depth-stencil state	***Default***
	Alternative
DepthEnable	***TRUE***

	FALSE
DepthWriteMask	***D3D10_DEPTH_WRITE_MASK_ALL*** (enables writing to the depth-stencil buffer)
	D3D10_DEPTH_WRITE_MASK_ZERO (disables writing to the depth-stencil buffer)
DepthFunc	***D3D10_COMPARISON_LESS*** (the test passes if the new data < existing data)
	D3D10_COMPARISON_NEVER (no depth test is performed)
	D3D10_COMPARISON_EQUAL (the depth test passes if the new data == existing data)
	D3D10_COMPARISON_LESS_EQUAL (the depth test passes if new data <= existing data)
	D3D10_COMPARISON_GREATER (the depth test passes if new data > existing data)
	D3D10_COMPARISON_NOT_EQUAL (the depth test passes if new data != existing data)
	D3D10_COMPARISON_GREATER_EQUAL (the depth test passes if new data >= existing data)
	D3D10_COMPARISON_ALWAYS (the depth test is always performed and always passes)
StencilEnable	***FALSE***
	TRUE
StencilReadMask	***D3D10_DEFAULT_STENCIL_READ_MASK***
StencilWriteMask	**D3D10_DEFAULT_STENCIL_WRITE_MASK**

Table 3.1 Default and alternative depth-stencil states.

Table 3.2 lists the **D3D10_DEPTH_STENCILOP_DESC** structure's possible stencil operations. These operations can be specified depending on the outcome of the stencil test. The **D3D10_DEPTH_STENCILOP_DESC** structure is a member of depth-stencil description which is specified using the **D3D10_DEPTH_STENCIL_DESC** structure.

Stencil Operation	Description
D3D10_STENCIL_OP_KEEP	Do not modify the existing stencil buffer data.
D3D10_STENCIL_OP_ZERO	Reset the stencil buffer data to zero.
D3D10_STENCIL_OP_REPLACE	Set the stencil buffer data to a reference value.
D3D10_STENCIL_OP_INCR_SAT	Increment the stored stencil buffer value by 1 (won't exceed the maximum clamped value).
D3D10_STENCIL_OP_DECR_SAT	Decrement the stored stencil buffer value by 1 (won't decrease below 0).
D3D10_STENCIL_OP_INVERT	Do a bitwise invert of the sorted stencil buffer data.
D3D10_STENCIL_OP_INCR	Increment the stored stencil buffer value by 1 (wrapping the result if required)
D3D10_STENCIL_OP_DECR	Decrement the stored stencil buffer value by 1

	(wrapping the result if required)

Table 3.2 Possible stencil operations.

A depth-stencil state (**depthstencilDesc**), specifying the details of the depth and stencil testing operations, is defined by first initialising the depth testing members, namely, **DepthEnable**, **DepthWriteMask** and **DepthFunc**:

```
D3D10_DEPTH_STENCIL_DESC depthstencilDesc;

depthstencilDesc.DepthEnable = true;
depthstencilDesc.DepthWriteMask = D3D10_DEPTH_WRITE_MASK_ALL;
depthstencilDesc.DepthFunc = D3D10_COMPARISON_LESS;
```

Following the above initialisation, the members required by the stencil test (**StencilEnable**, **StencilReadMask** and **StencilWriteMask**) must be initialised:

```
depthstencilDesc.StencilEnable = true;
depthstencilDesc.StencilReadMask = 0xFFFFFFFF;
depthstencilDesc.StencilWriteMask = 0xFFFFFFFF;
```

Next we have to setup the stencil operations for both back-facing and front-facing pixels via the **D3D10_DEPTH_STENCILOP_DESC** structure's members. For example, if **StencilFailOp** is set to **D3D10_STENCIL_OP_KEEP** and the stencil test fails then the current stencil buffer value is saved. Similarly, if **StencilDepthFailOp** is set to **D3D10_STENCIL_OP_DECR** with a failing stencil test, then the stencil buffer value is decremented by 1. Alternatively, the passing functions such as **StencilPassOp** only perform a stencil buffer operation on a passing stencil test and can have a different result depending on whether a pixel is back-facing or front-facing:

```
depthstencilDesc.BackFace.StencilFailOp =
                                        D3D10_STENCIL_OP_KEEP;
depthstencilDesc.BackFace.StencilDepthFailOp =
                                        D3D10_STENCIL_OP_DECR;
depthstencilDesc.BackFace.StencilPassOp =
                                        D3D10_STENCIL_OP_KEEP;
depthstencilDesc.BackFace.StencilFunc =
                                        D3D10_COMPARISON_ALWAYS;

depthstencilDesc.FrontFace.StencilFailOp =
                                        D3D10_STENCIL_OP_KEEP;
depthstencilDesc.FrontFace.StencilDepthFailOp =
                                        D3D10_STENCIL_OP_INCR;
depthstencilDesc.FrontFace.StencilPassOp =
                                        D3D10_STENCIL_OP_KEEP;
depthstencilDesc.FrontFace.StencilFunc =
```

```
                                        D3D10_COMPARISON_ALWAYS;
```

Now we simply have to set the depth stencil state to encapsulate all the above defined information for the pipeline stage determining the visible pixels. We do this using the **CreateDepthStencilState ID3D10Device** interface. This interface takes two parameters, the first a pointer to the depth-stencil state description (**D3D10_DEPTH_STENCIL_DESC**) structure and the second, the address of the depth-stencil state object (**ID3D10DepthStencilState**):

```
ID3D10Device * pID3D10Device;
ID3D10DepthStencilState * pDepthStencilState;

pID3D10Device->CreateDepthStencilState (depthstencilDesc,
                                        &pDepthStencilState);
```

With the depth stencil state set, we still have to create a depth-stencil buffer resource. This can be accomplished using a texture resource. Texture resources can be described as structured collections of data – specifically texture data. These structured data collections, as opposed to buffers, allow for the filtering of textures via texture samplers; with the exact filtering method determined by the texture resource type. Specifically, to create a depth-stencil buffer we require a texture resource (defined using the **ID3D10Texture2D** interface) consisting of a two-dimensional grid of texture elements (specified via the **D3D10_TEXTURE2D_DESC** structure describing a two-dimensional texture resource):

```
ID3D10Texture2D* pDepthStencilBuffer = NULL;
D3D10_TEXTURE2D_DESC depthResource;
```

The members of the texture resource (**D3D10_TEXTURE2D_DESC**) are initialised as follows, with the **BindFlags** member set to the **D3D10_BIND_DEPTH_STENCIL** enumeration to identify the texture resource as a depth-stencil buffer (refer to the DirectX SDK documentation (Microsoft, 2008) for a description of the **D3D10_TEXTURE2D_DESC** structure and each of its members):

```
depthResource.Width = backBufferSurfaceDescription.Width;
depthResource.Height = backBufferSurfaceDescription.Height;
depthResource.MipLevels = 1;
depthResource.ArraySize = 1;
depthResource.Format = pDeviceSettings ->
                                    d3d10.AutoDepthStencilFormat;
depthResource.SampleDesc.Count = 1;
depthResource.SampleDesc.Quality = 0;
depthResource.Usage = D3D10_USAGE_DEFAULT;
depthResource.BindFlags = D3D10_BIND_DEPTH_STENCIL;
depthResource.CPUAccessFlags = 0;
depthResource.MiscFlags = 0;
```

The **ID3D10Device** method, **CreateTexture2D**, is used to create a two-dimensional array – the depth-stencil buffer. This method takes three parameters where the first parameter is a pointer to the above defined two-dimensional texture resource structure (**D3D10_TEXTURE2D_DESC**), the second is a pointer to a texture subresource ('NULL' in our case) and the third is the address of a pointer to the specified texture (**pDepthStencilBuffer**):

```
pID3D10Device->CreateTexture2D(&depthResource, NULL,
                                &pDepthStencilBuffer);
```

The finial step in configuring depth and stencil functionality is to bind the previously defined depth and stencil data to the output-merger stage. The *output-merger* stage is the final pipeline step dealing with pixel visibility. This step controls pixel visibility by incorporating pixel shader data with depth and stencil testing results. We start by binding the depth stencil state, **pDepthStencilState**, to the output-merger stage using the **OMSetDepthStencilState** method. This method takes two parameters with the first being a pointer to the depth-stencil state interface (**pDepthStencilState**). This depth-stencil state interface was previously created using the **CreateDepthStencilState ID3D10Device** interface. The second parameter, an unsigned integer, is the reference value we are doing the depth-stencil test against:

```
pID3D10Device->OMSetDepthStencilState(pDepthStencilBuffer, 1);
```

Next the view mechanism is used to describe how the depth-stencil resource will be handled (viewed) by the pipeline. In this case we use a depth stencil view, thus defining the resource as a depth stencil. The **D3D10_DEPTH_STENCIL_VIEW_DESC** structure, given here, is used for this purpose and is contained within the DirectX 10 d3d10.h header file:

```
typedef struct D3D10_DEPTH_STENCIL_VIEW_DESC {
    DXGI_FORMAT Format;
    D3D10_DSV_DIMENSION ViewDimension;
    union
    {
        D3D10_TEX1D_DSV Texture1D;
        D3D10_TEX1D_ARRAY_DSV Texture1DArray;
        D3D10_TEX2D_DSV Texture2D;
        D3D10_TEX2D_ARRAY_DSV Texture2DArray;
        D3D10_TEX2DMS_DSV Texture2DMS;
        D3D10_TEX2DMS_ARRAY_DSV Texture2DMSArray;
    };
} D3D10_DEPTH_STENCIL_VIEW_DESC;
```

The first member, **Format**, controls the data resource interpretation and it can range from a typeless, unsigned-interger or signed-interger to floating-point format. The given code example uses the **DXGI_FORMAT_D32_FLOAT** format (a 32-bit floating-

point format). The second member, **ViewDimension**, is used to determine the depth-stencil access method. This member is set to the **D3D10_DSV_DIMENSION_TEXTURE2D** constant, indicating the depth-stencil resources access type as a two-dimensional texture (due to the depth-stencil resource being defined as a two-dimensional texture resource). Alternative constants include:

- **D3D10_DSV_DIMENSION_TEXTURE1D** (for a one-dimensional depth-stencil resource)
- **D3D10_DSV_DIMENSION_TEXTURE1DARRAY** (for accessing the depth-stencil resource as an array consisting of one-dimensional textures)
- **D3D10_DSV_DIMENSION_TEXTURE2DARRAY** (for accessing the depth-stencil resources as an array consisting of two-dimensional textures)
- **D3D10_DSV_DIMENSION_TEXTURE2DMS** (for a two-dimensional depth-stencil resource with multisampling support)
- **D3D10_DSV_DIMENSION_TEXTURE2DMSARRAY** (for accessing the depth-stencil resources as an array consisting of two-dimensional textures with multisampling support) and
- **D3D10_DSV_DIMENSION_UNKNOWN** (for depth-stencil resources where the access method is to be determined dynamically during depth-stencil view creation).

Only one member contained within the union are to be initialised. **Texture1D** is initialised by setting the **D3D10_TEX1D_DSV** structure's **MipSlice** member to an integer value when a one-dimensional texture is required as a depth-stencil view ('0' indicates the first mipmap level in the depth-stencil view). **Texture1DArray** specifies the texture and related mipmap level when a one-dimensional texture array is required as a depth-stencil view. This member is of the type **D3D10_TEX1D_ARRAY_DSV** and requires the initialisation of three members, namely, **MipSlice** (the depth-stencil view's mipmap level, with '0' indicating the first mipmap level in the depth-stencil view), **FirstArraySlice** (the texture, stored in the array, to use in the depth-stencil view) and **ArraySize** (the number of textures, stored in the array, to use in the depth-stencil view). Similarly, **Texture2D** is initialised by setting the **D3D10_TEX2D_DSV** structure's **MipSlice** member to an integer value when a two-dimensional texture is required as a depth-stencil view ('0' indicates the first mipmap level in the depth-stencil view). **Texture2DArray** specifies the texture and related mipmap level when a two-dimensional texture array is required as a depth-stencil view. This member is of the type **D3D10_TEX2D_ARRAY_DSV**, and just as with **Texture1DArray** requires the initialisation of three members, namely, **MipSlice** (the depth-stencil view's mipmap level, with '0' indicating the first mipmap level in the depth-stencil view), **FirstArraySlice** (the texture, stored in the array, to use in the depth-stencil view) and **ArraySize** (the number of textures, stored in the array, to use in the depth-stencil view). The final two members, **Texture2DMS** and **Texture2DMSArray**, are initialised when using a multisampled two-dimensional texture and a multisampled two-dimensional texture array as a depth-stencil respectively. The **D3D10_TEX2DMS_DSV** structure's **UnusedField_Nothing ToDefine** member can be initialised to any integer value with the

D3D10_TEX2DMS_ARRAY_DSV structure having two members, namely, **FirstArraySlice** (the texture, stored in the array, to use in the depth-stencil view) and **ArraySize** (the number of textures, stored in the array, to use in the depth-stencil view). The following code sample defines the depth stencil resource as a view:

```
D3D10_DEPTH_STENCIL_VIEW_DESC depthstencilviewDescription;

depthstencilviewDescription.Format = DXGI_FORMAT_D32_FLOAT;
depthstencilviewDescription.ResourceType =
                                    D3D10_RESOURCE_TEXTURE2D;

depthstencilviewDescription.Texture2D.FirstArraySlice = 0;
depthstencilviewDescription.Texture2D.ArraySize = 1;
depthstencilviewDescription.Texture2D.MipSlice = 0;
```

Following this, we simply have to create and bind the depth stencil view to the output-merger stage using the **CreateDepthStencilView** and **OMSetRenderTargets** **ID3D10Device** interfaces. The **CreateDepthStencilView** method, creating the depth-stencil view, takes three parameters, namely a pointer to an **ID3D10Texture2D** object (**pDepthStencilBuffer**) used for storing the resource data, a pointer to the **D3D10_DEPTH_STENCIL_VIEW_DESC** structure and the address of a pointer to an **ID3D10DepthStencilView** interface (**pDepthStencilView**) used for controlling the texture resource utilised during the depth-stencil test:

```
ID3D10DepthStencilView* pDepthStencilView;

pID3D10Device->CreateDepthStencilView(pDepthStencilBuffer
                            &depthstencilviewDescription,
                            &pDepthStencilView);
```

The **OMSetRenderTargets** method binds this depth stencil view to the output-merger stage. It takes three parameters, with the first identifying the number of render targets, the second a pointer to a render target view array, and the third a pointer to the to the **ID3D10DepthStencilView** interface. A render target is written to by the output-merger stage, containing the pixel colour information:

```
ID3D10RenderTargetView* pRenderTargetView;

pID3D10Device->OMSetRenderTargets(1, &pRenderTargetView,
                                    pDepthStencilView);
```

The **OMSetDepthStencilState** **ID3D10Device** interface is used to update the depth stencil state. This update is performed by setting the output-merger stage's depth-stencil state. The **OMSetDepthStencilState** method takes two parameters with the first parameter a pointer to an **ID3D10DepthStencilState** interface

(`pDepthStencilState`) and the second the reference value we are doing the depth-stencil test against:

```
pID3D10Device->OMSetDepthStencilState(pDepthStencilState, 0);
```

As an alternative to the above described process, we can initialise the depth and stencil states using the DirectX FX system. The FX system also allows us to view the contents of a stencil buffer, something not possible via the Direct3D 10 API. The final chapter's stencil shadow volume implementation makes use of FX techniques to set the stencil and depth testing states as well as to do the physical stencil rendering. It is, however, important to remember that the depth-stencil testing process is always conducted in the same manner, regardless of the implementation details. This complete depth testing process (used to determine the pixels positioned closest to the camera) and stencil testing process (controlling, via a mask, which pixels to update) are outlined in Figure 3.3 and Figure 3.4 respectively.

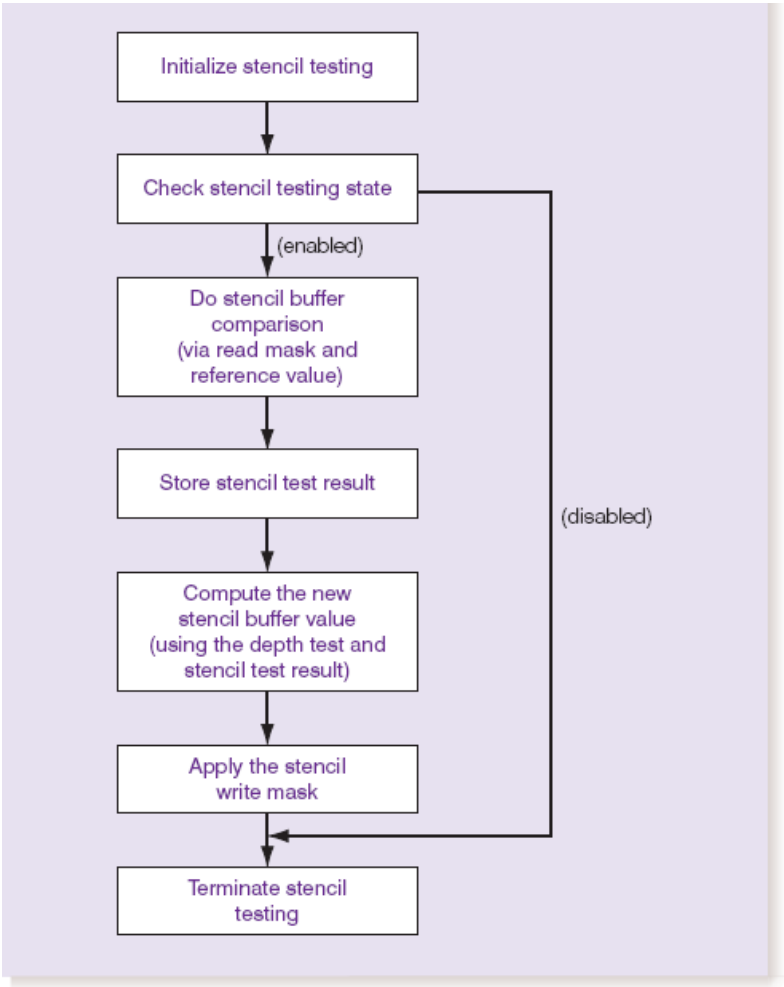


Figure 3.3 The stencil testing process.

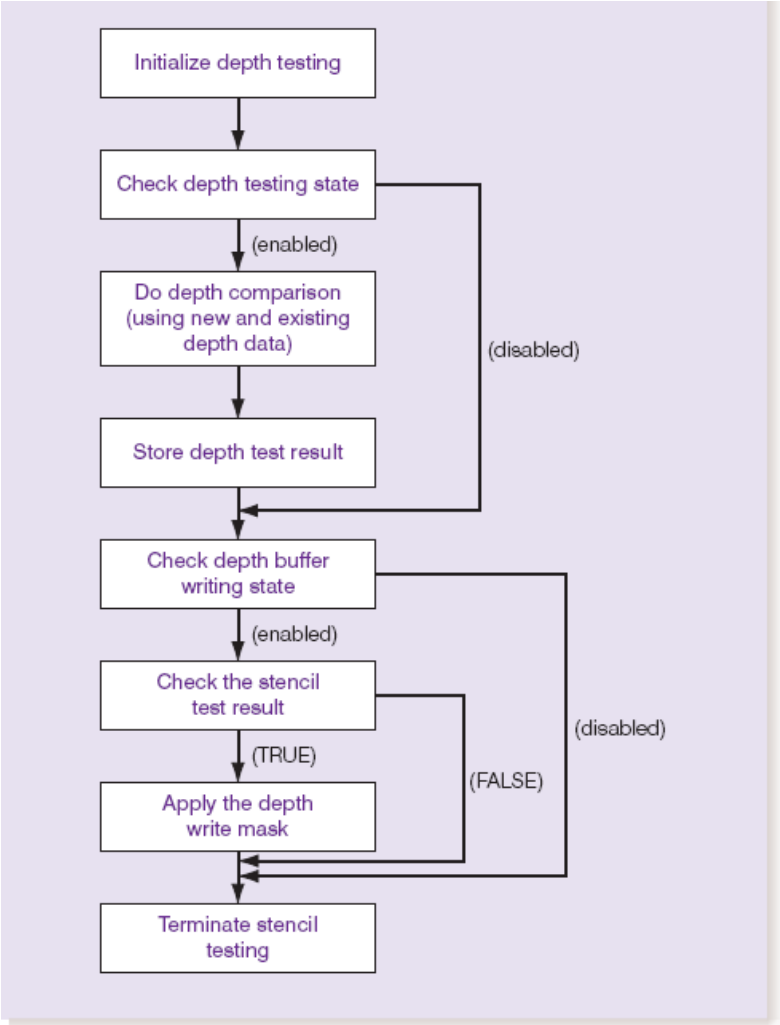


Figure 3.4 The depth testing process.

3.1.2 Implementing Stencil Shadow Volumes

The first step of a shadow volume implementation is to construct the shadow volume itself. This process starts with the calculation of silhouette edges followed by the generation of the shadow volume geometry. We use a shader to calculate the silhouette edges of an object with respect to a light source (we could alternatively calculate these edges on the CPU). The given geometry shader calculates the silhouette edges by determining the normal of each triangle face followed by the normals of the adjacent triangles. Thus, if the current triangle normal is facing the light source, with the adjacent triangle normal facing away, then we can flag their shared edge as a silhouette.

Our shader programs starts with the declaration of three structures for the storage of vertex and normal coordinate parameters.

```
struct VERTEXSHADER_INPUT
{
    float4 Loc : POSITION;
    float3 Norm : NORMAL;
};

struct PIXELSHADER_OUTPUT
{
    float4 Loc : SV_POSITION;
};

struct GEOMETRYSHADER_INPUT
{
    float4 Loc: POSITION;
    float3 Norm : NORMAL;
};
```

The first structure, **VERTEXSHADER_INPUT**, holds our vertex information as received from the Direct3D application and is used to pass input data to a vertex shader that transforms the input vertex position to clip space. It also transforms the input vertex normal to world space, finally returning the transformed vertex data via the **GEOMETRYSHADER_INPUT** structure:

```
/* vertex shader for sending the vertex data to the shadow
   volume geometry shader */
GEOMETRYSHADER_INPUT ShadowVertexShader(VERTEXSHADER_INPUT IN)
{
    GEOMETRYSHADER_INPUT output = (GEOMETRYSHADER_INPUT)0;

    /* transforms the input vertex position to world space */
    output.Loc = mul(float4(IN.Loc,1), WorldMatrix);

    /* transforms the input vertex normal to world space */
    output.Norm = mul(IN.Norm, (float3x3)WorldMatrix);

    return output;
}
```

Next we write a geometry shader to determine an object's silhouette edges using groups of vertices, each group consisting of two shared vertices and one adjacent vertex. This shader function also receives an un-normalised triangle normal (**normal**) as input; returning a **TriangleStream** containing the extruded shadow volume as a series of triangles. The shader starts by calculating the light vector pointing from the triangle towards the light source. This is followed by the calculation of the dot product between the triangle normal and the light vector. This dot product value is greater than '0' for triangles facing towards the light source. Following the initialisation of the shadow volume extrusion amount, **shadowExtrusionAmount**,

and bias, **shadowExtrusionBias** (for extending the shadow volume silhouette edges) we iterate through the adjacent triangles, calculating the silhouette edges and extruding the volumes out of the determined silhouettes. The geometry shader's final operation is to create the front- and back-cap of the newly defined shadow volume. Before listing this shader we need to look at a new input primitive type, **triangleadj**. This newly supported geometry shader type flags every other vertex as an adjacent vertex, in other words simplifying the work required to find the silhouette edges:

```
[maxvertexcount(18)]
void SilhouetteEdgeAndVolumeGS(
        triangleadj GEOMETRYSHADER_INPUT vertex[6],
        float3 normal,
        inout TriangleStream<PIXELSHADER_INPUT> ExtrudedVolume)
{
    /* determine the light vector from the triangle to light
       source */
    float lightVector = LightPosition - In[0].Loc;

    /* calculate the triangle normal */
    float triangleNormal = cross(In[2].Loc - In[0].Loc,
                                 In[4].Loc - In[0].Loc);

    /* calculate the dot product between the triangle normal
       and the light vector - if this value (the length of
       triangleNormal projected onto the lightVector) is
       greater than '0' then the triangle is facing the light
    */
    float3 projectionLength = dot(triangleNormal,
                                  lightVector);

    PIXELSHADER_OUTPUT Output;

    /* set the amount and bias to extrude the shadow
       volume from the silhouette edge */
    float shadowExtrusionAmount = 119.9f;
    float shadowExtrusionBias = 0.1f

    /* iterate through the adjacent triangles - where:
        - vertex[0], vertex[1] and vertex[6] are adjacent
        - vertex[2], vertex[3] and vertex[4] are adjacent
        - vertex[4], vertex[5] and vertex[0] are adjacent */
```

```
for(int i = 0; i < 6; i += 2)
{
   /* calculate the adjacency triangle normal */
   float triangleNormal = cross(vertex[i].Loc -
                                vertex[i+1].Loc,
                                vertex[i+2].Loc -
                                vertex[i+1].Loc);

   /* calculate the silhouette edges and extrude for
      triangles facing the light source */
   if(projectionLength > 0.0f)
   {
      float3 silhouette[4];

      /* extrude the silhouette edges */
      ////////////////////////////////////
      silhouette[0]= vertex[i].Loc +
                     shadowExtrusionBias *
                     normalize(vertex[i].Loc -
                               LightPosition);

      silhouette[1]= vertex[i].Loc + shadowExtrusionAmount*
                     normalize(vertex[i].Loc -
                               LightPosition);

      silhouette[2]= vertex[i+2].Loc + shadowExtrusionBias*
                     normalize(vertex[i+2].pos -
                               LightPosition);

      silhouette[3] = vertex[i+2].Loc +
                      shadowExtrusionAmount *
                      normalize(vertex[i+2].Loc -
                                LightPosition);

      /* create two new triangles for the extruded
         silhouette */
      Output.Loc=mul(float4(silhouette[v],1),ViewMatrix);

      //append shader-output data to an existing stream
      TriangleStream.Append(Output);
   }

   //end the current-primitive strip and start a new one
   TriangleStream.RestartStrip();
}
```

```
    /* create the front- and back-cap for the newly created
       triangles */

    //start with the nearest cap
    for(int k = 0; k < 6; k += 2)
    {
       float3 nearCapPosition = vertex[k].Loc +
                                shadowExtrusionBias *
                                normalize(vertex[k].Loc -
                                          LightPosition);

       Output.Loc = mul(float4(nearCapPosition,1), ViewMatrix);
       TriangleStream.Append(Output);
    }
    TriangleStream.RestartStrip();

    //now calculate the furthest cap
    for(int k = 4; k >= 0; k -= 2)
    {
       float3 farCapPosition =  vertex[k].Loc +
                                shadowExtrusionAmount *
                                normalize(vertex[k].Loc -
                                          LightPosition);

       Output.Loc = mul(float4(farCapPosition,1), ViewMatrix);
       TriangleStream.Append(Output);
    }
    TriangleStream.RestartStrip();
}
```

We can test whether a fragment is in shadow or not using either the previously discussed depth-fail or depth-pass technique. The chosen depth-stencil test can be implemented using native Direct3D 10 structures and functions as listed in section 3.1.1. The final step is to render the scene, resulting in the update of the pixels located inside the shadow volume and thus leading to the generation of shadowed regions.

3.2 The Shadow Mapping Algorithm

The shadow mapping evaluation environment is identical to our stencil shadow volume example's with only the shadow generation algorithm differing. Figure 3.5 illustrates this environment with a cast shadow.

Figure 3.5 Rendering a shadow by means of a shadow map (via one light source and three-dimensional mesh) – accurately cropped and skewed to fit the surrounding area.

The shadow mapping algorithm is implemented using the following method (described in Chapter 1):

1) Create the shadow map by rendering the Z-buffer with regard to the light's point of view.
2) Draw the scene from the viewer's point of view.
3) For each rasterised fragment, calculate the fragment's coordinate position with regard to the light's point of view.
4) Use the x- and y- coordinates of step 3's translated vertex to index the shadow Z-buffer.
5) Do the depth comparison test, if the translated vertex's depth value (the z-value of step 3's translated vertex) is greater that the value stored in the shadow Z-buffer, then the fragment is shadowed, else it is lit.

3.2.1 Implementing Shadow Maps

Shadow mapping, unlike shadow volumes, doesn't require any geometry-processing or mesh generation. We can thus, when using shadow maps, maintain a high level of performance regardless of the scene's geometric complexity.

The first step of a shadow mapping implementation is to render the scene from the light source's point of view. This is a trivial operation since the scene is already rendered to begin with – we simply have to reposition our camera. Following this, we can create the shadow map using the following call (Direct3D 8 or better):

```
pD3DDevice->CreateTexture(textureWidth, textureHeight,
                          1, D3DUSAGE_DEPTHSTENCIL,
                          D3DFMT_D24S8, D3DPOOL_DEFAULT,
                          &pTexture);
```

Basic shadow mapping in Direct3D is dependent on modification of the existing texture format – so we will, in essence, be making use of Direct3D's render-to-texture capabilities. These render-to-texture capabilities allow us to render directly to the shadow map texture [Everitt et al, 2001].

With the shadow map created, we simply have to texture it onto the scene. This operation requires a projection transformation followed by the alignment of shadowed and screen pixels. This alignment often causes changes in a pixel's screen size (which is, as mentioned, responsible for aliasing errors).

Also, Direct3D's **`SetRenderTarget`** operation requires the creation of a colour surface as it combines the depth surface with the colour surface. Everitt et al (2001) explains the actual rendering process best: "you render from the point of view of the light to the shadow map you created, then set the shadow map texture in a texture stage and set the texture coordinates in that stage to index into the shadow map at *(s/q, t/q)* and use the depth value *(r/q)* for the comparison." *(s/q, t/q)* is the fragment's location within the depth texture with *(r/q)* the window-space depth of the fragment in relation to the light source's frustum. The following texture matrix can be used post-projection to setup our texture coordinates [Everitt et al, 2001]:

```
float fOffsetX = 0.5f + (0.5f / fTexWidth);
float fOffsetY = 0.5f + (0.5f / fTexHeight);

D3DXMATRIX texScaleBiasMat(0.5f, 0.0f, 0.0f, 0.0f,
                           0.0f, -0.5f, 0.0f, 0.0f,
                           0.0f, 0.0f, fZScale, 0.0f,
                           fOffsetX, fOffsetY, fBias, 1.0f );
```

`fZScale` is set to $(2^{\text{bit-planes}} - 1)$ with **`fBias`** set to any small arbitrary value.

All that remains now is to do the actual shadow test. We basically compare the depth of the window-space fragment against the depth texture fragment location. The result of this test can be either one (indicating a lit pixel) or zero to indicate a shadowed one. The easiest way to implement the shadow mapping process is via basic HLSL pixel and vertex shaders:

```
/* vertex shader for shadow mapping vertex processing */
void VertexShadow(float3 Normal : NORMAL,
                  float4 Pos : POSITION,
                  out float2 depth : TEXCOORD0,
                  out float4 outputPos : POSITION)

{
     /* calculate the projected coordinates */
     outputPos = mul(Pos, viewMatrix);
     outputPos = mul(outputPos, projMatrix);
```

```
      /* store the z- and w-coordinates using the available
         coordinates*/
      depth.xy = outputPos.zw;
}

/* shadow map pixel shader – processes shadow map pixels */

void PixelShadow(out float4 colour : COLOR,
                 float2 depth : TEXCOORD0)
{
      /* the depth is actually x/y
      colour = Depth.x / Depth.y;
}
```

3.3 Hybrid and Derived Approaches

We now present the implementation details for a number of hybrid stencil shadow volume/shadow mapping approaches. Several documented enhancements are also highlighted.

3.3.1 Shadow Volume Reconstruction from Depth Maps

The first approach that should be mentioned is McCool's (2000) shadow volume reconstruction through the use of depth maps. Michael McCool (2000) describes this approach as follows: "Current graphics hardware can be used to generate shadows using either the shadow volume or shadow map techniques. However, the shadow volume technique requires access to a representation of the scene as a polygonal model, and handling the near plane clip correctly and efficiently is difficult; conversely, accurate shadow maps require high-precision texture map data representations, but these are not widely supported… [The algorithm is] a hybrid of the shadow map and shadow volume approaches which does not have these difficulties and leverages high-performance polygon rendering. The scene is rendered from the point of view of the light source and a sampled depth map is recovered. Edge detection and a template-based reconstruction technique are used to generate a global shadow volume boundary surface, after which the pixels in shadow can be marked using only a one-bit stencil buffer and a single-pass rendering of the shadow volume boundary polygons. The simple form of our template-based reconstruction scheme simplifies capping the shadow volume after the near plane clip."

McCool's hybrid algorithm is implemented as follows (McCool, 2000):

1) Render the shadow map by drawing the scene from the light source's point of view.

2) Draw the scene from the viewer's point of view.
3) Reconfigure the frame buffer by clearing the stencil buffer and disabling writing to the colour and depth buffers.
4) Enable depth testing.
5) Set the stencil buffer to toggle when a shadow polygon fragment passes the depth test.
6) Render the shadow volume.
    a. The shadow volume is constructed from the shadow map's depth coordinates (z[x, y]) – these coordinates are translated to world space and projected through the same viewing transformation as the rest of the scene.
    b. The shadow volume's front- and back faces are rendered simultaneously as it is unnecessary to distinguish between them.
7) Generate shadow volume cap polygons (to ensure proper enclosure of the shadow volume).
8) Render the darkened pixels (where the stencil bit is set to 1)
9) Render the shadow using one of the following modes:
    a. Ambient mode – the stencil buffer isn't used and the scene is re-rendered using ambient illumination (masked to modify all pixels in shadow).
    b. Black mode – a single black polygon is drawn over the entire scene and all pixels in shadow are blackened.
    c. Composite mode – a semi-transparent black polygon is drawn over the entire scene and all pixels in shadow are darkened.

The most interesting part of McCool's algorithm is perhaps its use of multiple shadow maps. This is due to single shadow maps being limited to a field of view. Multiple shadow maps can be used to cast shadows omnidirectionally. McCool's approach assigns each spatial area a specific shadow map (the viewing frustum is adjusted to render extra depth samples around the edges when rendering the shadow maps).

3.3.2 Hybrid Algorithm for the Efficient Rendering of Hard-edged Shadows

Another interesting hybrid approach is the one developed by Chan and Durand (2004). Their approach, as previously mentioned, combines the strengths of shadow maps and shadow volumes to produce a hybrid algorithm for the efficient rendering of pixel-accurate hard-edged shadows. Their method uses a shadow map to identify pixels located near shadow discontinuities, using the stencil shadow volume algorithm only at these pixels. This approach ensures accurate shadow edges while actively avoiding the edge aliasing artefacts associated with standard shadow mapping as well as the high fillrate consumption of standard shadow volumes. The algorithm, in their own words "relies on a hardware mechanism for rapidly rejecting non-silhouette pixels during rasterization. Since current graphics hardware does not directly provide this mechanism, we simulate it using available features related to occlusion culling and show that dedicated hardware support requires minimal changes to existing technology".

The hybrid algorithm of Chan and Durand (2004) is implemented as follows:

1) Create the shadow map by placing the camera at the light source and rendering the nearest depth values to a buffer.
2) Find all the shadow silhouette pixels by rendering the scene from the viewer's point of view.
    a. Transform each test sample to light space and compare its depth against the four nearest depth samples from the shadow map.
        i. If the comparison results disagree, then we classify the sample as a silhouette pixel else we classify it as a non-silhouette pixel (which is in turn shaded according to the depth comparison test).
    b. Perform z-buffering to prepare the depth buffer for the shadow volume drawing in step 3.
3) Render the shadow volumes using the depth-fail stencil shadow volume algorithm.
4) Render and shade all the pixels with stencil values equal to zero.

The following pixel shader, as given by Chan and Durand (2004), illustrates the silhouette detection process:

```
void main (out half4 color : COLOR,
           half diffuse : COL0,
           float4 uvProj : TEXCOORD0,
           uniform sampler2D shadowMap)
{
     // Use hardware's 2x2 filter: 0 <= v <= 1.
     fixed v = tex2Dproj(shadowMap, uvProj).x;

     // Requirements for silhouette pixel: front-facing and
     // depth comparison results disagree.
     color = (v > 0 && v < 1 && diffuse > 0) ? 1 : 0;
}
```

The exact silhouette detection process is based on the depth comparison between image samples and the four nearest depth samples as found in the shadow map. If this comparison returns a "0" or "1", then we can say that the depth comparison results agree (the pixel is thus not a silhouette pixel). A disagreeing result indicates a silhouette pixel.

3.3.3 Elimination of various Shadow Volume Testing Phases

Thakur et al (2003), as previously mentioned, developed a discrete algorithm for improving the Heidmann original. Their algorithm was primarily based on the elimination of various testing phases which resulted in an overall performance gain when compared to the original. Thakur et al (2003) formally describe this technique

as follows: “[it] does not require (1) extensive edge/edge intersection tests and intersection angle computation in shadow polygon construction, or (2) any ray/shadow-polygon intersection tests during scan-conversion. The first task is achieved by constructing ridge edge (RE) loops, an inexact form of silhouette, instead of the silhouette. The RE loops give us the shadow volume without any expensive computation. The second task is achieved by discretizing the shadow volume into angular spans. The angular spans, which correspond to scan lines, are stored in a lookup table. This lookup table enables us to mark the pixels that are in shadow directly, without the need of performing any ray/shadow-polygon intersection tests. In addition, the shadow on an object is determined on a line-by-line basis instead of a pixel-by-pixel basis. The new technique is efficient enough to achieve real time performance, without any special hardware, while being scalable with scene size”.

The hybrid algorithm of Thakur et al (2003) is implemented as follows:

1) Construct a Lookup Table by:
   a. Finding the ridge edges.
   b. Connecting the ridge edges to form loops.
   c. Determining the angular coordinates (r,θ,ϕ) of all vertices positioned on ridge edge loops.
   d. Identifying the vertices with local peaks in θ (ridge edge loops are sliced along θ with local peaks being specific points in the loop).
   e. Appending all the points of edges to the lookup table until a minimum in θ is reached (by starting from the identified peaks).
   f. Inserting the hidden edges in the lookup table.
   g. Performing a pair-wise sorting of all entries in the lookup table (in terms of ϕ).
2) Perform scan conversion and generate a query at each point (x, y, z) to determine whether the point’s in shadow or not (this is done for each scan line) – see Figure 3.6.
3) Calculate the Maximum Run Length (the distance on a scan line for which θ stays the same).
4) Depending on the return value of step 2’s function, create or don’t create a shadow up to **`nextX`** or **`x+MRL`** (which ever comes first).
5) Perform the subsequent shadow query.

It’s interesting to note the contrast between Thakur et al’s algorithm as compared to traditional stencil shadow volume methods, that is; shadow determination stops when the first instance of a shadow is found (the actual shadow is a logical OR of all cast shadows). It is thus unnecessary to traverse the entire list; an insight that results in an overall performance increase. Shadow volumes conversely require the interception and counting of each and every shadow polygon.

```
Convert input x, y, z to r, θ and φ
If table entry at θ exists
   nextX = END
   For all pairs (φ_i, φ_j) of table entry
        If (φ_i ≤ φ ≤ φ_j AND r_i ≤ r)
           nextX = xEquivalentOf(φ_j)
           return TRUE
        Else if(φ < φ_i)
           tempX = xEquivalentOf(φ_i)
           If (tempX < nextX)
              nextX = tempX
   return FALSE
Else
   nextX = END
   return FALSE
```

Figure 3.6 The query function as given by Thakur et al (2003).

3.3.4 Shadow Volumes and Spatial Subdivision

Another noteworthy solution, as presented in Rautenbach et al (2008), combines the depth-fail stencil shadow volume algorithm with spatial subdivision – an approach researched and developed as part of the author's postgraduate studies. This approach, as a unification that results in real-time frame rates for rather complex scenes, primarily deals with statically lit environments and is an apt shadowing model and improvement over the traditional Heidmann (1991) algorithm.

This algorithm enhances the current depth-fail and depth-pass stencil shadow volume algorithms by enabling more efficient silhouette detection, thus reducing the number of unnecessary surplus shadow polygons. It also includes a technique for the efficient capping of polygons, thus effectively handling situations where shadow volumes are being clipped by the point-of-view near clipping plane.

Crucial to this implementation is the Octree data structure. Relying on this data structure, an Octree algorithm sorts the collections of polygons that make up the shadow volumes into a specific visibility order. This order is pre-determined by the viewpoint. Our approach uses the Octree to calculate the shadow volume unification by traversing the tree in a front-to-back order, thus in effect subdividing the surface (endpoint) polygons for each element/object.

The approach basically uses the resulting Octree to calculate the shadow volume unification by traversing the tree in a front-to-back order, thus in effect subdividing the surface (endpoint) polygons for each element/object. Chapter 4 presents this approach in more detail.

3.4 Summary

In the chapter we started by presenting a basic stencil shadow volume benchmarking program consisting of a relatively simple static cubic environment, a movable/interchangeable three-dimensional mesh object and a variable number of light sources. Next we introduced the stencil buffer as a buffer used for controlling the rendering of selected pixels. The associated per-pixel test, namely stencilling, was also looked at in detail. We then investigated the depth-stencil testing process followed by a discussion detailing the shadow volume implementation (with specific focus being on the shadow volume construction process and calculation of silhouette edges).

Following this we looked at a shadow mapping evaluation environment as well as the implementation of shadow maps. We then dealt with the implementation details of various hybrid approaches such as McCool's shadow volume reconstruction using depth maps, Chan and Durand's hybrid algorithm for the efficient rendering of hard-edged shadows, Thakur et al's elimination of various shadow volume testing phases and Rautenbach, Pieterse and Kourie's shadow volumes, hardware extensions and spatial subdivision approach.

The next chapter focuses on the critical analysis and benchmarking of the presented shadow rendering algorithms – the data to be used by our fuzzy-based expert system for the real-time selection of these algorithms (see Chapter 5).

Chapter 4

Benchmarking of Shadow Algorithms

Chapter 4 presents the critical analysis and detailed benchmarking of the previously discussed shadowing techniques. The chapter also looks at the performance of several optimised shadow volume and shadow mapping routines. The knowledge base of our expert system draws heavily on these experimental results.

In this chapter we will investigate:

- Benchmarking Mechanism
- Evaluation Criteria
- Scalability
- Construction Complexity
- Rendering Accuracy and Detail
- Instruction Set Utilisation
- Evaluation of the Basic Stencil Shadow Volume Algorithm
- Evaluation of the Basic Hardware Shadow Mapping Algorithm
- Evaluation of Shadow Volume Reconstruction from Depth Maps
- Evaluation of a Hybrid Algorithm for the Efficient Rendering of Hard-edged Shadows
- Elimination of various Shadow Volume Testing Phases
- Performance Analysis of Shadow Volumes and Spatial Subdivision

4.1 Benchmarking Mechanism

Benchmarking entails running a computer program with the aim of assessing its performance. This action is normally hardware-centric and intended to measure the performance of numerous subsystems and/or execution routines. We use such a system to evaluate the previously discussed shadow rendering algorithms. This benchmarking system basically functions as a plug-in where real-time performance data are streamed to a file-based database for post-processing and analysis.

Each evaluated algorithm was plugged into a base Direct3D 10 rendering framework. This framework, specifically written for this study, offers conventional first-person perspective input control (via DirectInput and later XInput for Xbox 360 controllers), mesh loading and support for shader model 4.0 shaders (via Microsoft's High Level Shader Language). The algorithms were benchmarked without special effects such as normal and displacement mapping or the later added motion blur effect (see accompanying video). The test framework also supports the dynamic placement, movement and elimination of light sources. The critical analysis was performed via scripted camera movement and light source additions – this was done not only to ensure consistent testing, but also to ease future validation and replication of results.

4.2 Evaluation Criteria

We now present a set of criteria which was used to evaluate the depth-fail stencil shadow volume, shadow mapping and hybrid shadow generation techniques. The given evaluation criteria were selected with the aim of assessing the relationship between shadow rendering quality and performance – in turn allowing us to isolate key algorithmic weaknesses and possible bottleneck areas. Table 4.1 lists the proposed evaluation criteria in the first column, indicating in parenthesis whether its focus is on quality, performance or both. The second column provides motivation for the criterion's inclusion.

Evaluation Criteria	Motivation
Scalability ***(performance)***	Evaluating the performance of an algorithm based on the intensifying complexity of the rendered scene allows for the identification of algorithmic limits and the maximum threshold for scene and model complexity. (Analyse the overall performance impact due to an increase in the number of light sources and the shadow casting model's polygonal complexity).
Construction Complexity	Assessing the total number of clock

(performance)	cycles and the number of clock cycles for every routine of the shadow rendering algorithm allows us to identify not only the processor intensive operations, but also the areas where the algorithmic improvements can lead to the biggest overall performance gain.
Rendering Accuracy and Detail ***(quality)***	Determining whether a shadow is cropped and/or skewed properly, and accurately projected onto other models and surfaces, allows for the evaluation of shadow rendering quality.
Instruction Set Utilisation ***(performance/quality)***	Comparing a standard C/Direct3D 10 implementation to one extended through the utilisation of Intel's SSE and AMD's 3DNow! instruction set allows for the evaluation of maximized parallelism (offered by these instruction sets) versus conventionally executed routines.

Table 4.1 Evaluation criteria.

4.3 Experimentation and Results

We will now evaluate a number of shadow rendering algorithms, specifically the stencil shadow volume algorithm, the shadow mapping algorithm and a number of hybrid approaches such as McCool's shadow volume reconstruction using depth maps, Chan and Durand's hybrid algorithm for the efficient rendering of hard-edged shadows, Thakur et al's elimination of various shadow volume testing phases and Rautenbach et al's shadow volumes, hardware extensions and spatial subdivision approach. Evaluation of the previously mentioned scan-line polygon projection as well as Blinn's shadow polygons are excluded due to these algorithms being historic in nature.

4.3.1 Basic Stencil Shadow Volume Algorithm

To gather the necessary results, we implemented the depth-fail stencil shadow volume algorithm for a number of scenes. In each case, the scene was a relatively simple cubic environment featuring a single movable 3D model and a variable number of light sources. The 3D models utilised are those provided as samples by the Microsoft DirectX SDK. The test system had the following configuration:

NVIDIA GeForce™8800 GTX 768 MB GDDR3 (Video Card),

AMD Athlon™3000+ (Processor),
2.0GB (Memory),
1280x1024 (Screen Resolution).

Scalability

In order to accurately benchmark the scalability of the shadow volume algorithm, we render a somewhat simple scene consisting of a relatively modest polygon model (599 faces) and one light source. Figure 4.1 illustrates this scene with the rendered shadow and constructed shadow volume. Figures 4.2, 4.3 and 4.4 illustrate the shadows and constructed shadow volumes for the other models. Increasing the number of light sources will lead to additional shadows as shown in Figure 4.5.

Figure 4.1 (a) Shadow generated by the “tiger” model (1 light source).
(b) Constructed shadow volume.

Figure 4.2 (a) Shadow generated by the “car” model (1 light source).
(b) Constructed shadow volume.

Figure 4.3 (a) Shadow generated by the “shapes” model (1 light source).
(b) Constructed shadow volume.

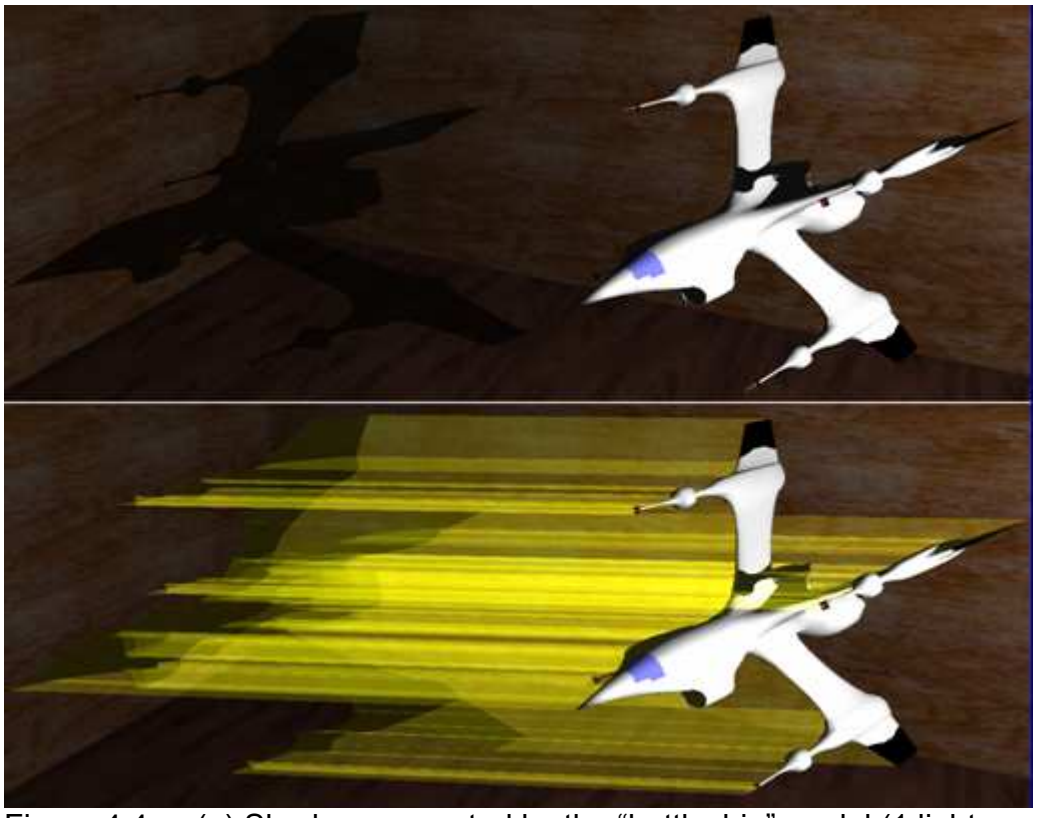


Figure 4.4 (a) Shadow generated by the “battleship” model (1 light source).
(b) Constructed shadow volume.

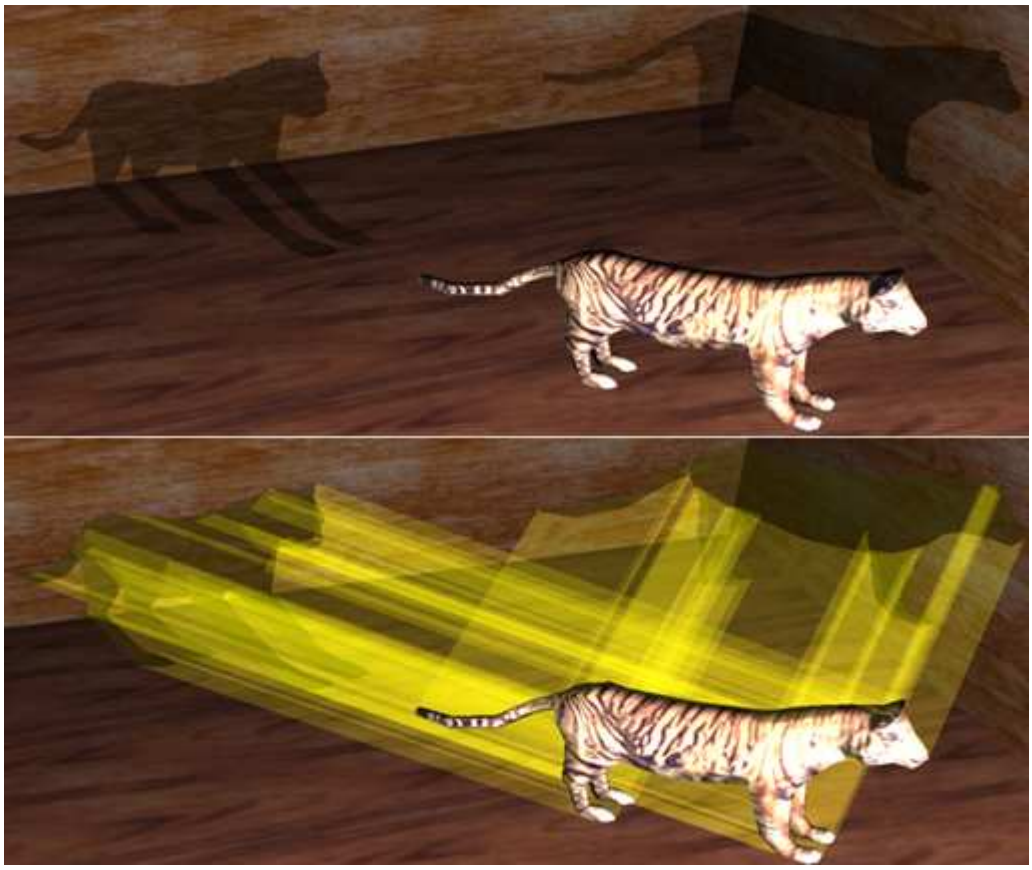


Figure 4.5 (a) Shadows generated by the “tiger” model (2 light sources).
(b) Constructed shadow volumes.

Following this initial rendering, we systematically increase the number of light sources while varying the model complexity. Figure 4.6 summarises the resulting performance data. By analysing these results, it is clear that both the number of light sources and mesh complexity have an influence on the performance of the stencil shadow volume algorithm. The primary bottleneck is in fact the shadow volume construction process. Increasing the number of light sources results in an additional shadow volume being created for each light source. Also, the greater the number of faces, the greater the number of silhouette edges considered during the shadow volume construction process.

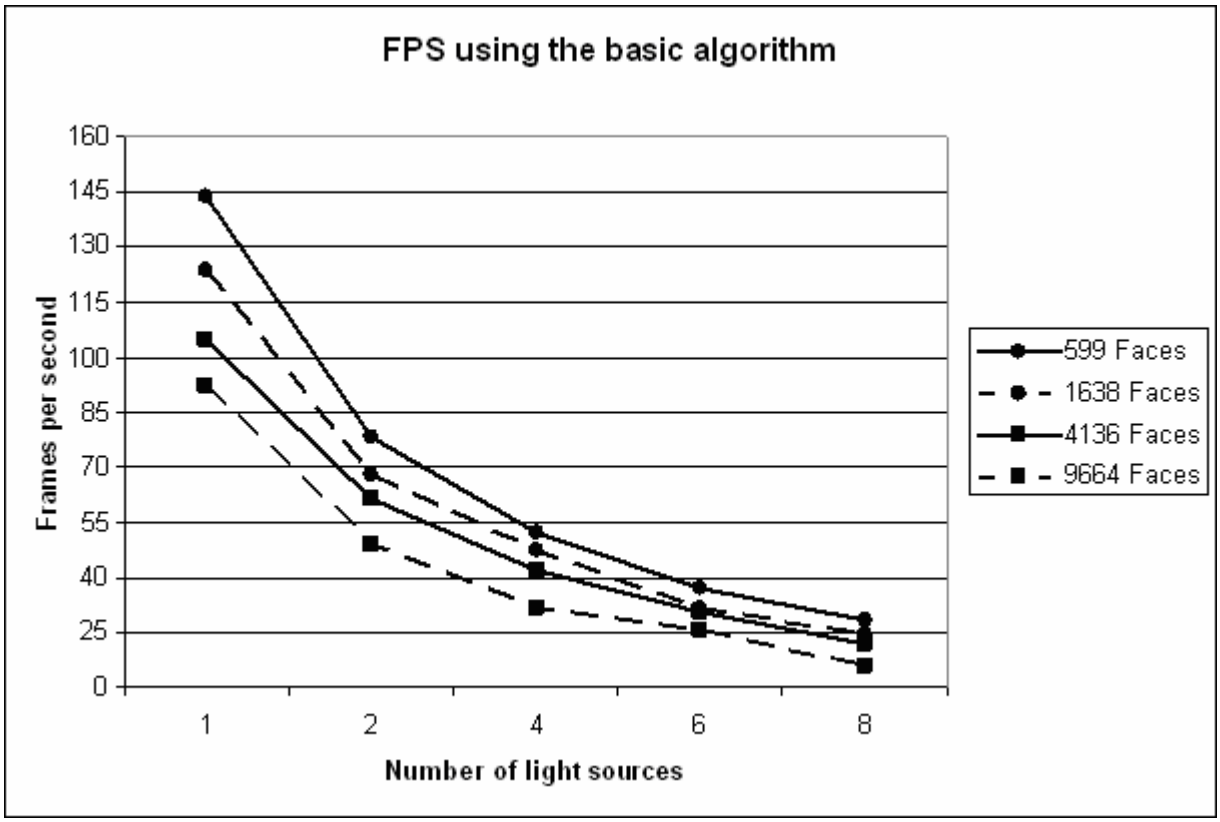


Figure 4.6 The correlation between the number of light sources and FPS rendering performance for polygonal meshes with varying number of faces using the basic stencil shadow volume algorithm.

Construction Complexity

Assessing the total number of clock cycles and the number of clock cycles for every routine of the shadow volume construction process allows us to isolate the processor intensive operations and possible areas of improvement. The results shown in Figure 4.7 compare the processor dependence of our 599-face 'tiger' mesh. These tests were executed on the same system as the one used for scalability testing. We also employed an Intel Pentium 4 'Northwood' 2.8 GHz – based test system (with all its hardware components corresponding to that of our AMD test machine). The data obtained from this machine is also shown in Figure 4.7. These results will be used when comparing the standard C/Direct3D implementation to the utilisation of Intel's SSE2 and AMD's 3DNow! instruction sets.

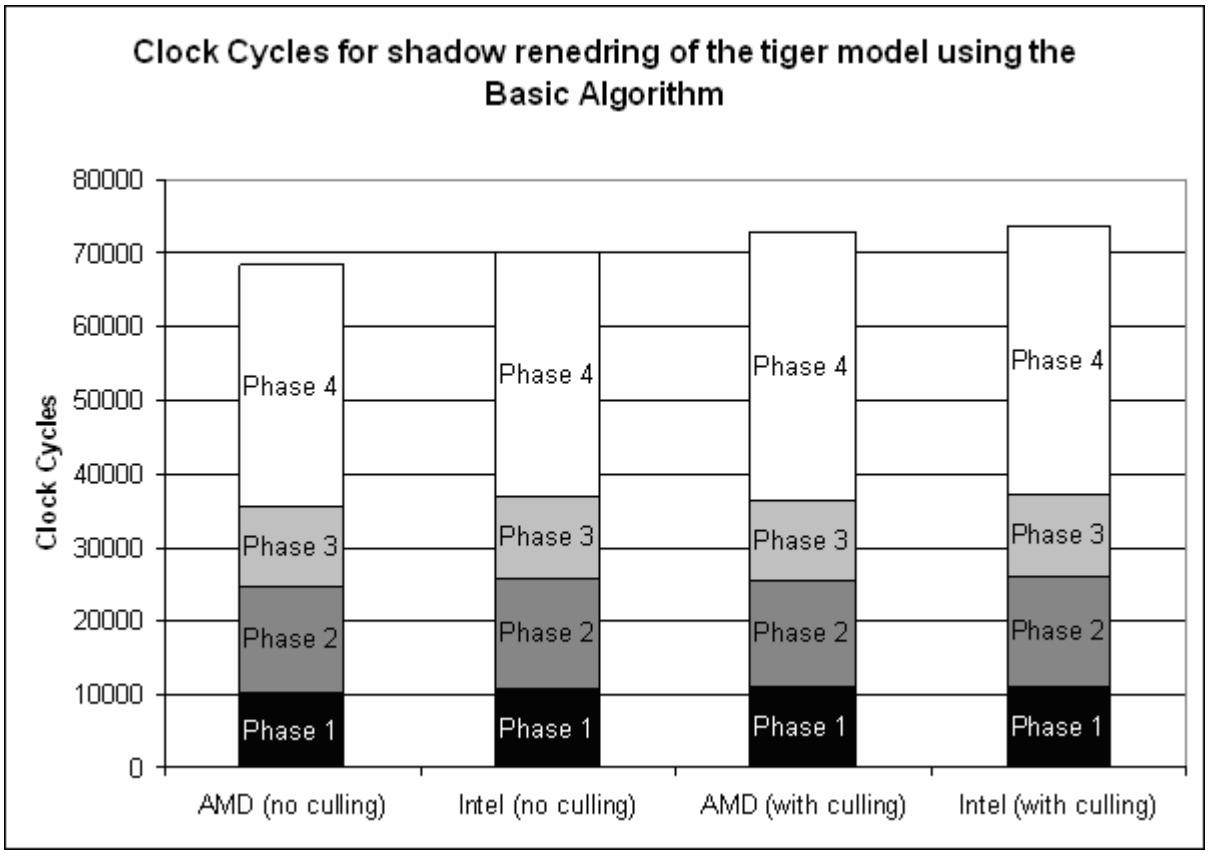


Figure 4.7 Comparing clock cycle iterations with and without culling for shadow volume construction of the tiger mesh (599 faces) using different hardware configurations.

We investigated the number of clock cycle iterations for each of the shadow volume construction phases. The first phase in this process is to calculate the number of light source facing triangles. The second phase involves construction of the silhouette polygons (also referred to as the silhouette plane polygons); with the third phase responsible for the construction of the shadow volume's capping triangles. The fourth phase is the actual construction of the shadow volume. We also modified this phase to construct shadow volumes while at the same time culling polygons located outside of the volume. We can similarly modify the first shadow volume construction phase to calculate the number of light source facing triangles with the culling of polygons outside of the volume. As can be seen in Figure 4.7, the difference between the two hardware configurations is minimal. The culling of polygons located outside of the volume involves more work initially, thus resulting in some performance loss.

We repeated the same experiment on the AMD machine using higher polygon models. As can be seen in Figure 4.8, the total number of clock cycles increases drastically when the number of faces increase, but the relation between the work done during the different phases does not differ significantly. The increasing clock cycle measurements are mostly due to the numerous conditional branches executed during the shadow volume creation process. Translating and/or rotating a mesh also impacts heavily on the clock cycle count due to the recalculation of the light source facing polygons and subsequently the shadow volume. Calculation of the number of light source facing polygons is the primary performance bottleneck and the only

foreseeable performance enhancement would be to execute all the current sequential-conditional code in parallel – the section on "Instruction Set Utilisation" deals with this in detail.

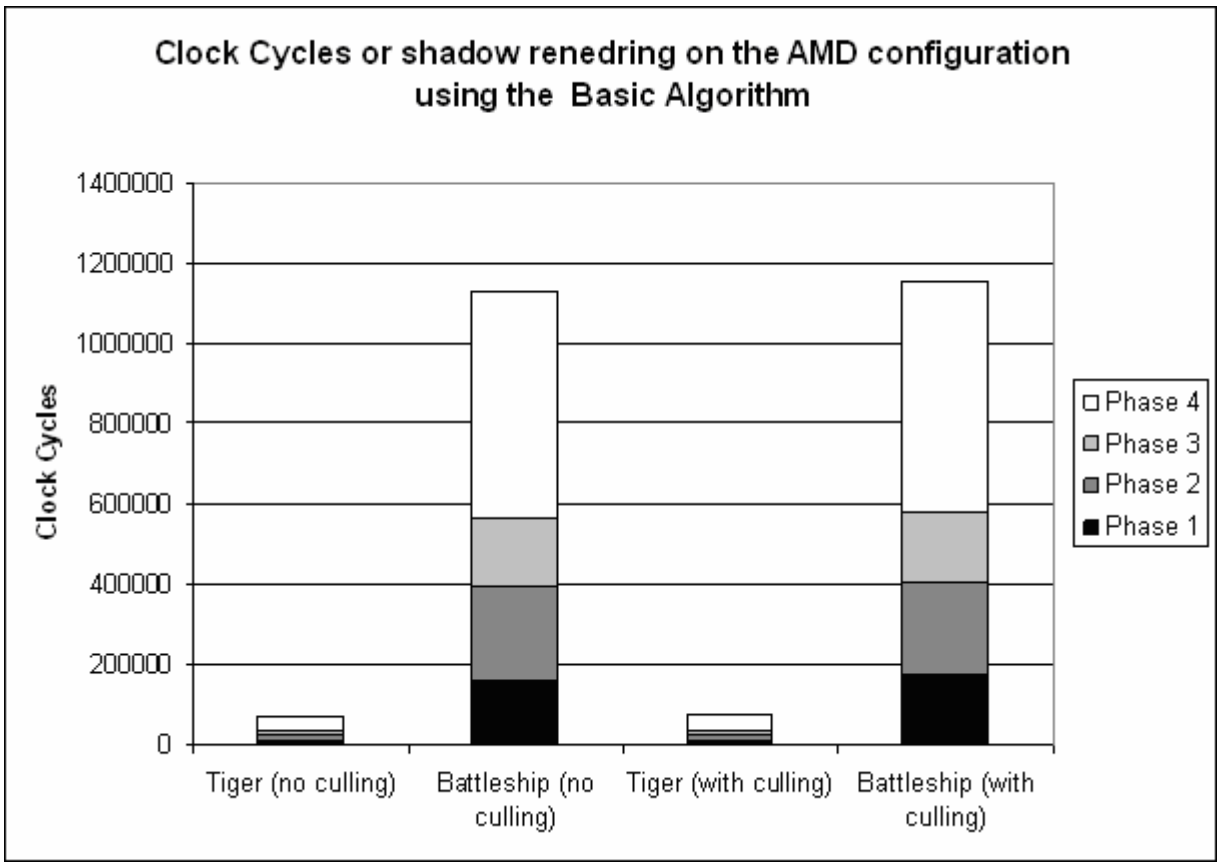


Figure 4.8 Comparing clock cycle iterations for shadow volume construction of the tiger (599 faces) and the battleship (9664 faces) – shown here with and without culling respectively.

Rendering Accuracy and Detail

The rendering accuracy and detail of projected shadows can be determined by way of visual observation. A model is translated and rotated within a scene and the detail of the projected shadow is investigated. Stencil shadow volumes, being per-pixel based, are by default of high quality. Figure 4.9 and 4.10 show a properly cropped and skewed shadow.

Figure 4.9 A properly cropped and skewed shadow.

Figure 4.10 Close-up showing pixel-perfect quality edges.

Although the generated shadow is of high quality, the shadow volume algorithm suffers from some artefacts in situations where the mesh casts shadows onto itself, as shown in Figure 4.11.

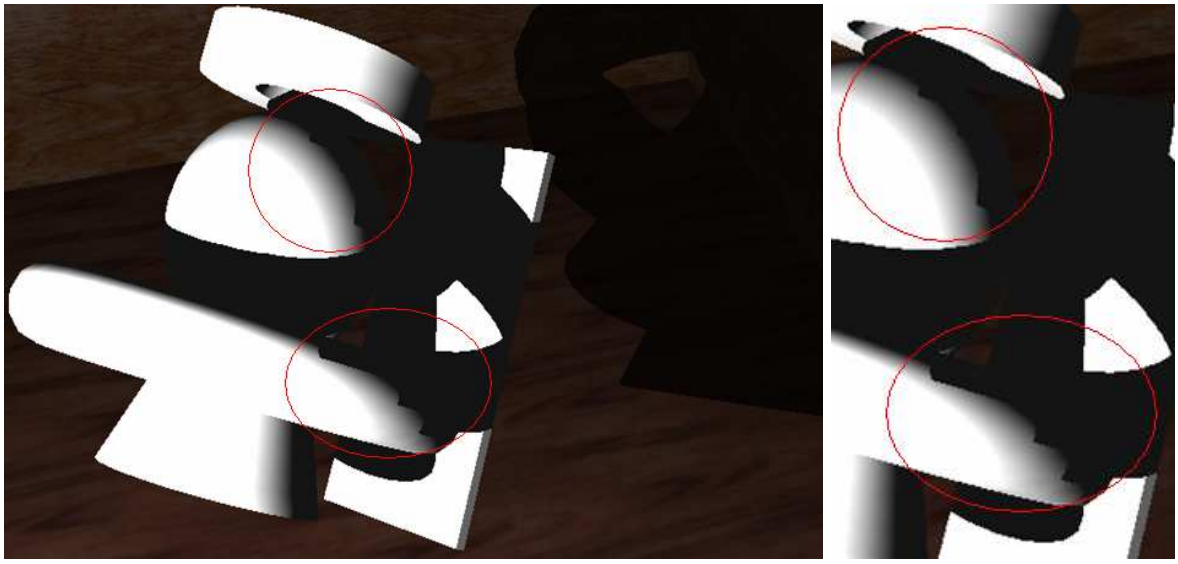


Figure 4.11 The encircled areas show the artefacts near the silhouette edges.

These commonly encountered and frequently discussed artefacts are the result of the shadow volume's lighting computation phase. An object's faces are rendered as shadowed or lit depending on whether a face normal points towards the light source or not. Thus, an entire face will be rendered as shadowed although only part of it is actually in shadow (something the algorithm doesn't consider). Shadow mapping doesn't suffer from this and these artefacts can only be reduced by increasing the mesh complexity – thus resulting in less noticeable artefacts near the silhouette edges.

Instruction Set Utilisation

Comparing a standard C/Direct3D implementation to the utilisation of Intel's SSE2 and AMD's 3DNow! instruction sets allow for the evaluation of maximised parallelism (as offered by these instruction sets) versus conventionally sequentially executed routines. The 3D Now! instruction set is a multimedia extension created by AMD for the improvement of vector processing and floating-point-intensive tasks as required by graphic-intensive and multimedia applications (AMD, 2000). Intel's SSE is similar in nature and stands for Streaming Single Instruction, Multiple Data Extensions. It is based on the principle of carrying out multiple computations with a single instruction in parallel (Intel, 2002). The SEE instruction set (specifically SSE2 found on the Pentium 4 architecture) also adds 64-bit floating point and 8/16/32-bit integer support. As can be seen in Figure 4.12, there is significant gain especially in the first phase. It is also noticeable that the second phase (i.e. the construction of the silhouette polygons) did not lend itself to any gain. The difference between the utilised hardware offerings is negligible.

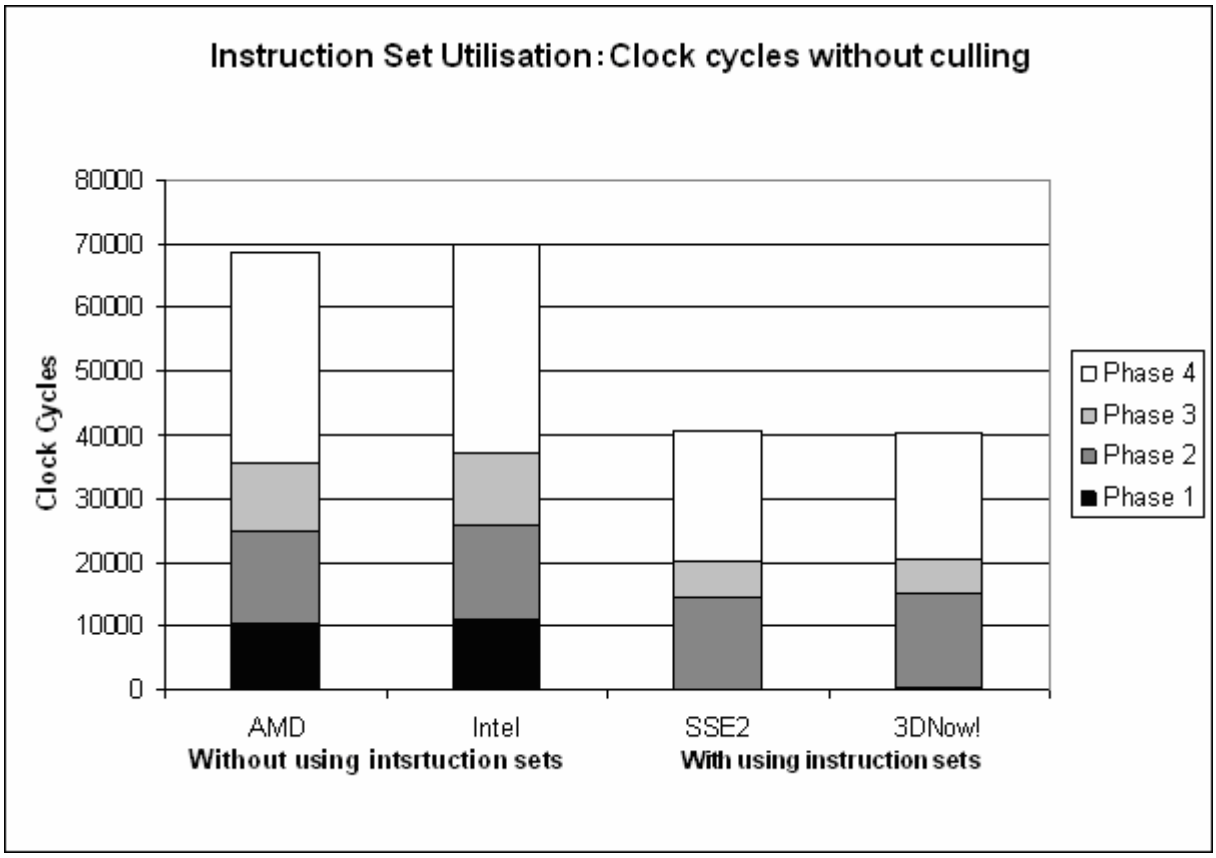


Figure 4.12 Comparing Clock cycle gains when using hardware instruction sets.

Figure 4.13 illustrates the comparison between pure SSE2 vs. 3DNow! Performance data and the results obtained by combining these instruction sets with culling.

To understand the concept of maximised parallelism, consider the following for-loop used to count the number of facing polygons:

```
int facing_triangles = 0;

for(int n = 0; n < numberOfTriangles; n++)
{
  if(triangleIsFacing(n))
  {
     facing_triangles = facing_trangles + 1;
  }
}
```

The given loop can be subdivided and parallelised via either SSE or 3DNow!, with each individual loop processing a portion of the triangle faces. This parallelised face counting results in a performance increase from 10841 clock cycles to 149 clock cycles for the SSE2 implementation and 10224 to 157 clock cycles for the 3DNow! implementation.

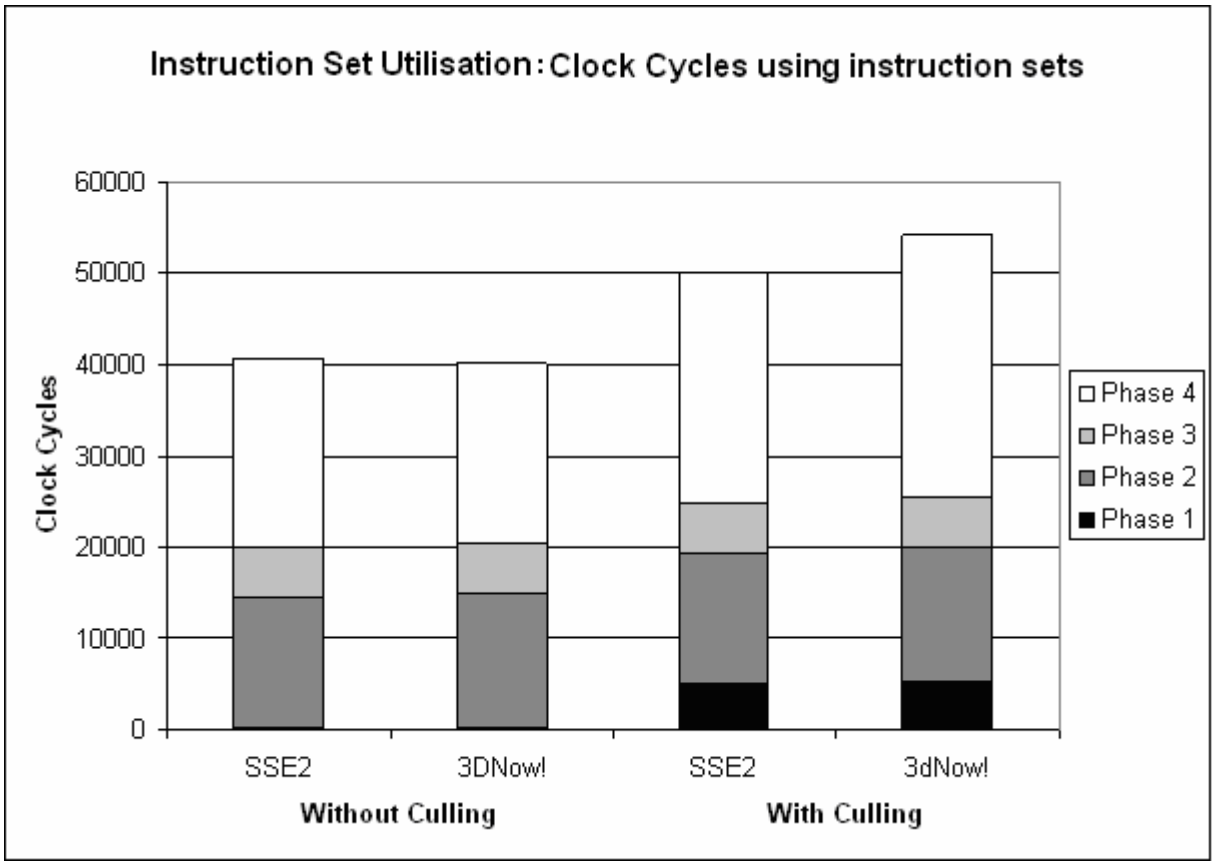


Figure 4.13 Comparing clock cycle results of SSE2 vs. 3DNow! with and without the use of culling.

Partitioning and parallelising these loops via SSE2 or 3DNow! lead to substantial performance increases (about 42%). The slight architectural differences between the Intel and AMD instruction sets lead to slightly varying results. The AMD architecture barely outperforms the Intel architecture when surfaces outside of the light volume aren't culled. One reason for this could be the AMD processor's clock speed - the Athlon64 3000+ operates at a substantially lower frequency than its Intel counterpart (2.8GHz). The AMD processor rating system (3000+ in this case) indicates that the processor is comparable to a 3.0GHz processor although operating at roughly 1.8GHz. This rating system is the result of AMD processors executing a greater number of instructions during each clock cycle than Intel's offerings (Rau, 2002). It is, however; patently clear that usage of either SSE or 3DNow! results in considerable performance gains.

4.3.2 Basic Hardware Shadow Mapping Algorithm

To gather the necessary results, we implemented the shadow mapping algorithm for a number of scenes. In each case, the scene was a relatively simple cubic environment featuring a single movable 3D model and a variable number of light sources. The 3D models utilised are those provided as samples by the Microsoft DirectX SDK. The test system had the following configuration:

NVIDIA GeForce^{TM}8800 GTX 768 MB GDDR3 (Video Card),

AMD Athlon™3000+ (Processor),
2.0GB (Memory),
1280x1024 (Screen Resolution).

Scalability

In order to accurately benchmark the scalability of the shadow mapping algorithm, we render a somewhat simple scene consisting of a relatively modest polygon model (599 faces) and one light source. Figure 4.14 shows this scene with the rendered shadow. Figures 4.15, 4.16 and 4.17 illustrate the shadows for the other models. Increasing the number of light sources will lead to additional shadows as shown in Figure 4.18.

Figure 4.14 Shadow generated by the “tiger” model (1 light source).

Figure 4.15 Shadow generated by the “car” model (1 light source).

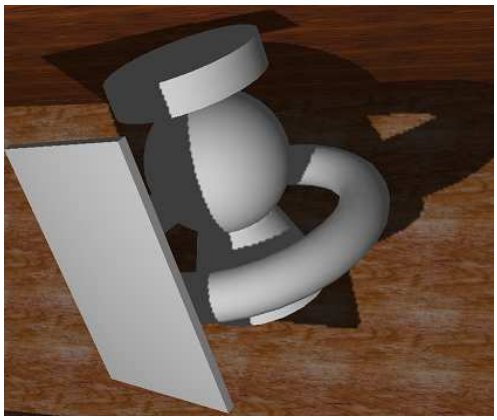


Figure 4.16 Shadow generated by the “shapes” model (1 light source).

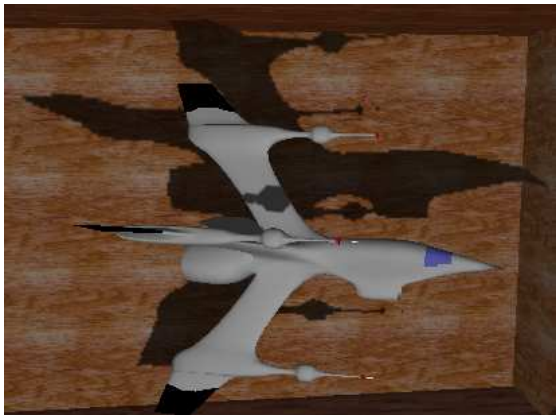


Figure 4.17 Shadow generated by the “battleship” model (1 light source).

Figure 4.18 Shadows generated by the “tiger” model (2 light sources).

Following this initial rendering, we systematically increase the number of light sources while varying the model complexity. Figure 4.19 summarises the resulting performance data. Increasing the number of light sources has a clear influence on the performance of the shadow mapping algorithm. The mesh complexity is, however, a less important factor when computing shadow maps due to the absence of any silhouette detection. The resolution of the shadow map has the biggest influence on the overall rendering speed. The technique is clearly less accurate than stencil shadow volumes. It is also much faster due to the stencil buffer not being utilised and no shadow volume polygons being rendered.

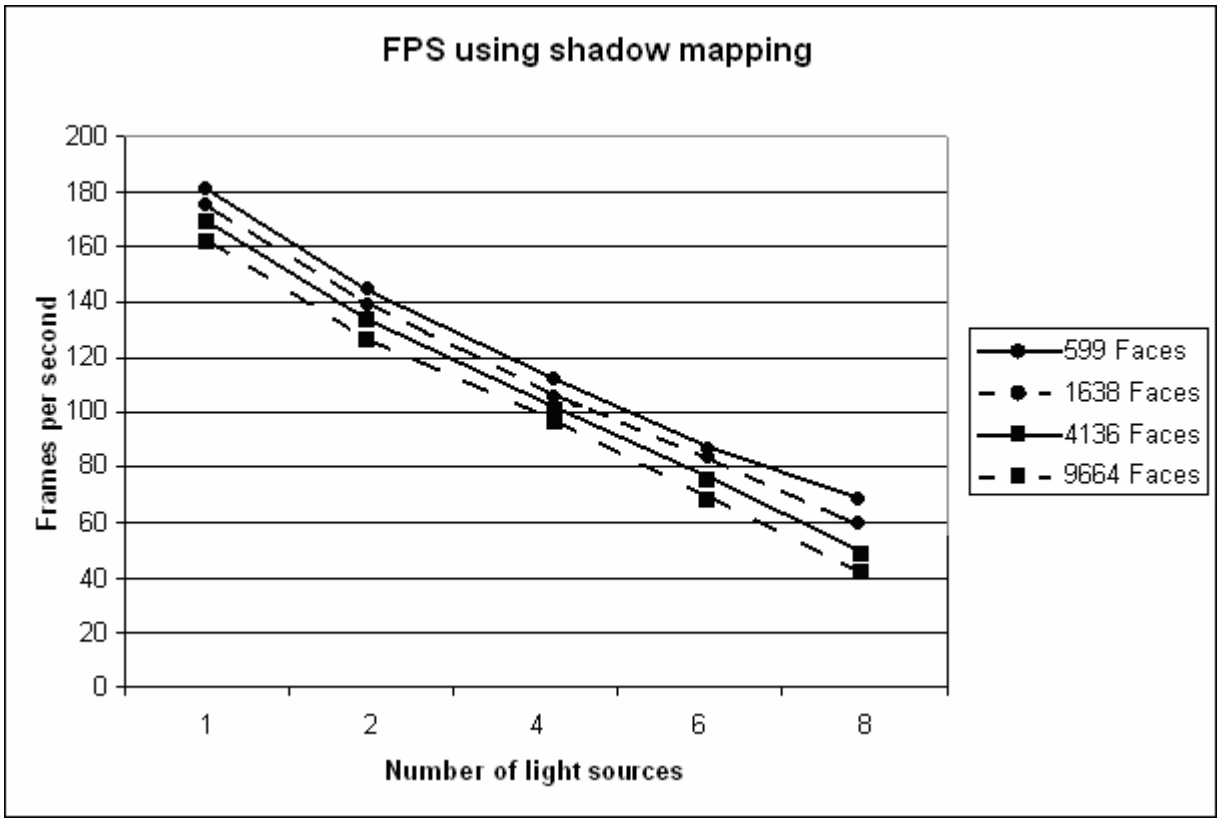


Figure 4.19 The correlation between the number of light sources and FPS rendering performance for polygonal meshes with varying number of faces using the basic hardware shadow mapping algorithm.

Construction Complexity

Shadow mapping, as previously mentioned, doesn't require any silhouette detection. There is also no need to calculate the number of light source facing triangles or to construct any silhouette polygons. The previous shadow volume construction complexity analysis looked at the clock cycles involved in the construction of shadow volume capping triangles as well as the construction of the shadow volume – these steps, and the related critical analysis, are thus no longer applicable.

Shadow mapping simply projects the Z-buffer with regard to the point of view of some light source followed by the scene being drawn from the viewer's point of view. The only real computationally intensive part is the calculation of the fragment coordinate position for every rasterised fragment based on the light's point of view. These fragment coordinates are then simply translated to Z-buffer indexes. Z-buffer indexes are subsequently used to determine whether the fragment's in shadow or not (using the previously described depth comparison test). The advent of the NVIDIA GeForce™ 3 series GPUs greatly improved shadow mapping via the addition of three hardware extensions, specifically the **`SGIX_depth_texture`**, **`GL_ARB_shadow_ambient`** and **`SGIX_shadow`** OpenGL extensions as well as a special Direct3D texture format (Everitt et al, 2002).

Rendering Accuracy and Detail

The rendering accuracy and detail of projected shadows can be determined by way of visual observation. A model was translated and rotated within a scene and the detail of the projected shadow was investigated. Shadow mapping, as previously mentioned, suffers from aliasing errors due to the use of a projection transformation mapping shadowed pixels to screen pixels, often causing changes in a pixel's screen size. This is a direct result of the Z-buffer algorithm's use of point sampling. The rendered shadow's edges are often jagged due to point sampling errors occurring during the calculation phase of the shadow Z-buffer. These errors are further amplified when accessing the shadow Z-buffer for the projection of pixels onto the shadow Z-buffer map. The only way of minimising the visibility of a shadow's jagged edges is to implement some form of pre-filtering and to use very large (high resolution) shadow maps. Figure 4.20 shows a properly cropped and skewed shadow.

Figure 4.20 A properly cropped and skewed shadow.

Instruction Set Utilisation

Shadow mapping, as opposed to stencil shadow volumes, doesn't require the CPU for the creation of shadow geometry or for the counting of facing triangles. Consequently, the performance gains resulting from the use of Intel's SSE2 or AMD's 3DNow! instruction sets are trivial (as opposed to conventional sequentially executed routines). We decided that it would be redundant to conduct experiments to confirm this.

4.3.3 Shadow Volume Reconstruction from Depth Maps (McCool)

Michael McCool (2000) proposed shadow volume reconstruction using depth maps. We were interested in determining the performance advantage of this algorithm over the traditional approach. To gather the necessary results, we implemented McCool's algorithm for a number of scenes. In each case, the scene was a relatively simple cubic environment featuring a single movable 3D model and a variable number of light sources. The 3D models utilised are those provided as samples by the Microsoft DirectX SDK. The test system had the following configuration:

NVIDIA GeForce™8800 GTX 768 MB GDDR3 (Video Card),
AMD Athlon™3000+ (Processor),
2.0GB (Memory),
1280x1024 (Screen Resolution).

Scalability

In order to accurately benchmark the scalability of McCool's algorithm, we render a somewhat simple scene consisting of a relatively modest polygon model (599 faces) and one light source. Figure 4.21 shows a properly cropped and skewed shadow generated using this algorithm.

Figure 4.21 Shadow generated using McCool's (2000) algorithm.

Following this initial rendering, we systematically increase the number of light sources while varying the model complexity. Figure 4.22 summarises the resulting performance data. By analysing these results, it is clear that both the number of light sources and mesh complexity have an influence on the performance of McCool's algorithm. McCool's (2000) algorithm has a slight performance advantage over the traditional approach without any loss in shadow quality – this is primarily due to elimination of the dot product computation previously utilised to find the silhouette edges of an object. The algorithm extrudes the silhouette edges via use of a depth map and vision technique. This results in an overall performance gain when compared to the traditional stencil shadow volume algorithm.

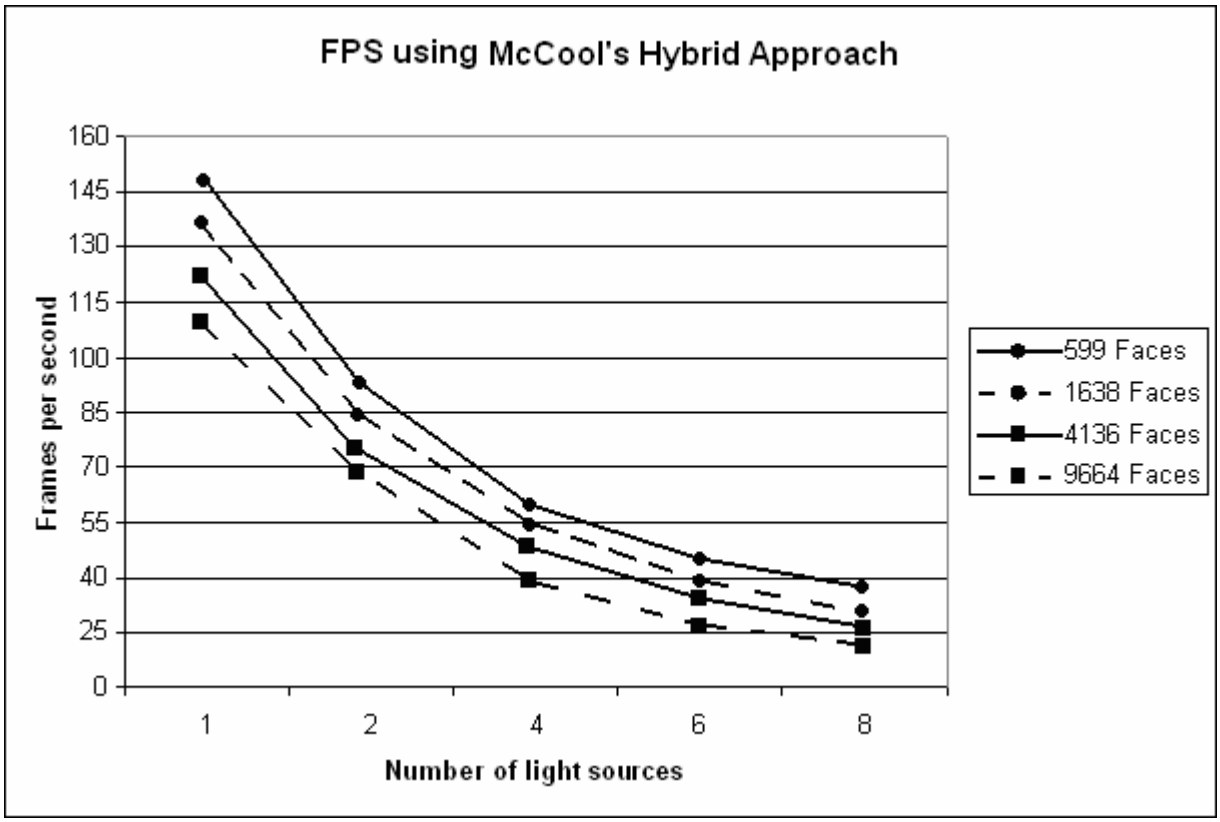


Figure 4.22 The correlation between the number of light sources and FPS rendering performance for polygonal meshes with varying number of faces using McCool's hybrid approach.

Construction Complexity

Assessing the total number of clock cycles and the number of clock cycles for every routine of McCool's shadow construction process allows us to isolate the processor intensive operations and possible areas of improvement. The results shown in Figure 4.23 compare the processor dependence of our 599-face 'tiger' mesh. These tests were executed on the same system as the one used for scalability testing. We also employed an Intel Pentium 4 'Northwood' 2.8 GHz – based test system (with all its hardware components corresponding to that of our AMD test machine). The data obtained from this machine is also shown in Figure 4.23. These results will be used when comparing the standard C/Direct3D implementation to the utilisation of Intel's SSE2 and AMD's 3DNow! instruction sets.

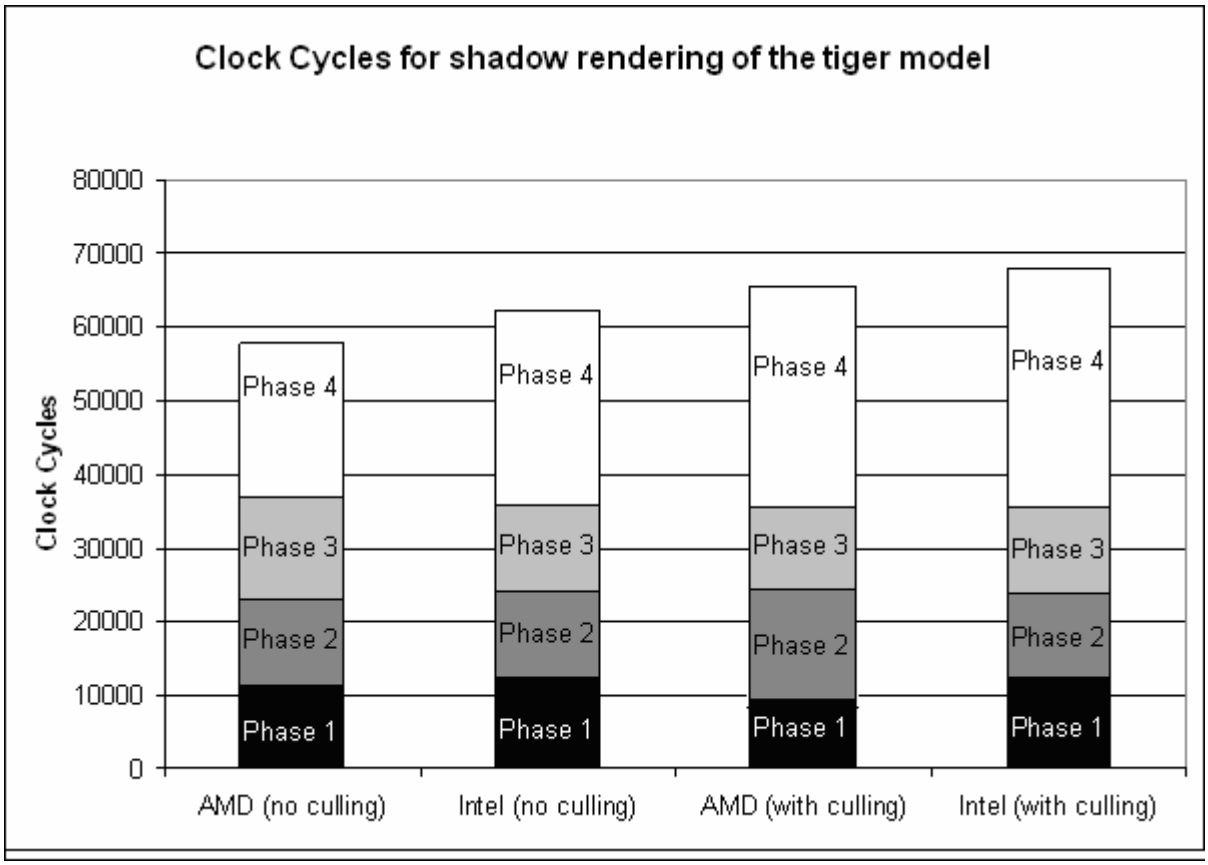


Figure 4.23 Comparing clock cycle iterations with and without culling for the tiger mesh (599 faces) using McCool's algorithm with different hardware configurations.

We investigated the number of clock cycle iterations for each of the shadow volume construction phases. The first phase in this process is to calculate the number of light source facing triangles. The second phase involves construction of the silhouette polygons (also referred to as the silhouette plane polygons) using the shadow map's depth coordinates (z[x, y]); with the third phase responsible for the construction of the shadow volume's capping triangles. The fourth phase is the actual construction of the shadow volume. We can also modify this phase to construct shadow volumes while at the same time culling polygons located outside of the volume. We can similarly modify the first shadow volume construction phase to calculate the number of light source facing triangles with the culling of polygons outside of the volume. As can be seen in Figure 4.23, the difference between the two hardware configurations is minimal. The culling of polygons located outside of the volume involves more work initially, thus resulting in some performance loss.

We repeated the same experiment on the AMD machine using higher polygon models. As can be seen in Figure 4.24, the total number of clock cycles increases drastically when the number of faces increase, but the relation between the work done during the different phases does not differ significantly. The increasing clock cycle measurements are mostly due to the numerous conditional branches executed during the shadow volume creation process. Translating and/or rotating a mesh also impacts heavily on the clock cycle count due to the recalculation of the light source facing polygons, the shadow map and subsequently the shadow volume.

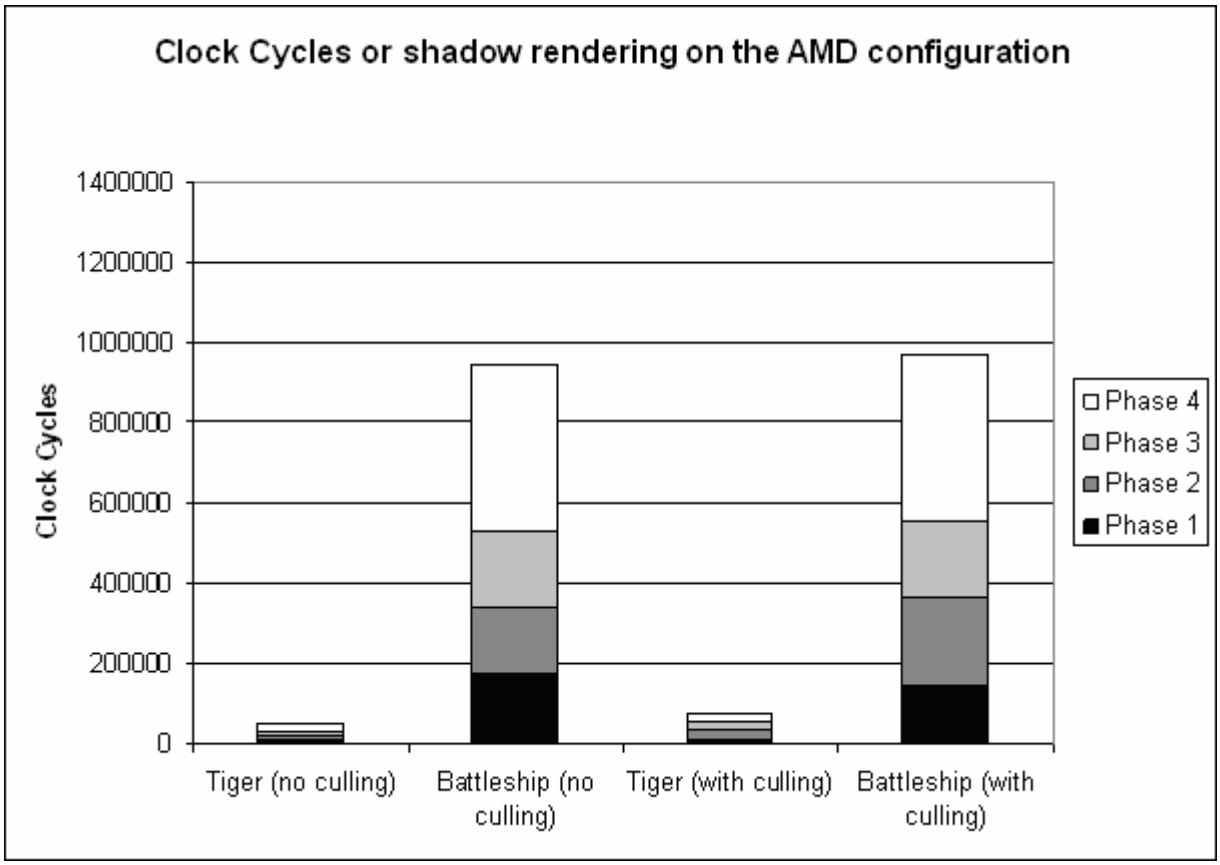


Figure 4.24 Comparing clock cycle iterations with McCool's algorithm for the tiger (599 faces) and the battleship (9664 faces) – shown here with and without culling respectively.

Rendering Accuracy and Detail

The rendering accuracy and detail of projected shadows can be determined by way of visual observation. A model is translated and rotated within a scene and the detail of the projected shadow is investigated. Shadow quality is heavily dependent on the depth map's resolution (compare the jaggedness of the shadow in Figure 4.20, rendered via a low resolution depth map, with Figure 4.25's smooth edges) and shadow edges can easily be anti-aliased (a technique used to reduce image distortion artefacts – jaggedness, known as aliasing) without any need to recapture the shadow map (McCool, 2000). Figure 4.25 shows a properly cropped and skewed shadow.

Figure 4.25 A properly cropped and skewed shadow.

Instruction Set Utilisation

McCool's algorithm, as stencil shadow volumes, requires the CPU for the creation of shadow geometry and for the counting of facing triangles. We will thus observe nearly identical performance gains by utilising Intel's SSE2 or AMD's 3DNow! instruction set (as opposed to conventional sequentially executed routines). There is significant gain especially in the first phase. It is also noticeable that the second phase did not lend itself to any gain. The difference between the utilised hardware offerings is negligible.

To understand the concept of maximised parallelism, consider the same for-loop used to count the number of facing polygons:

```
int facing_triangles = 0;

for(int n = 0; n < numberOfTriangles; n++)
{
  if(triangleIsFacing(n))
  {
     facing_triangles = facing_trangles + 1;
  }
}
```

The given loop can be subdivided and parallelised via either SSE or 3DNow!, with each individual loop processing a portion of the triangle faces. This parallelised face counting results in a performance increase from 8325 clock cycles to 98 clock cycles for the SSE2 implementation and 9005 to 103 clock cycles for the 3DNow! implementation.

All the other shadow volume construction phases use similar loops with either the culling or the filling of triangle indices added. Partitioning and parallelising these loops via SSE2 or 3DNow!, as previously discussed, lead to substantial performance increases. The AMD architecture, as observed in our other experiments, barely

outperforms the Intel architecture when surfaces outside of the light volume aren't culled. It is, however, clear that usage of either SSE or 3DNow! results in considerable performance gains (bar charts aren't given as the performance gains are nearly identical to those given in section 4.3.1).

4.3.4 Algorithm for the Efficient Rendering of Hard-edged Shadows (Chan and Durand)

Eric Chan and Frédo Durand (2004) combine the strengths of shadow maps and shadow volumes to produce a hybrid algorithm for the efficient rendering of pixel-accurate hard-edged shadows. To gather information about the relative performance gains of this approach, we implemented Chan and Durand's (2004) hybrid algorithm for a number of scenes. In each case, the scene was a relatively simple cubic environment featuring a single movable 3D model and a variable number of light sources. The 3D models utilised are those provided as samples by the Microsoft DirectX SDK. The test system had the following configuration:

NVIDIA GeForce™8800 GTX 768 MB GDDR3 (Video Card),
AMD Athlon™3000+ (Processor),
2.0GB (Memory),
1280x1024 (Screen Resolution).

Scalability

In order to accurately benchmark the scalability of Chan and Durand's (2004) hybrid algorithm, we start by rendering a somewhat simple scene consisting of a relatively modest polygon model (599 faces) and one light source.

Following this initial rendering, we systematically increase the number of light sources while varying the model complexity. Figure 4.26 summarises the resulting performance data. By analysing these results, it is clear that both the number of light sources and mesh complexity have an influence on the performance of Chan and Durand's algorithm. The actual performance gains are extremely hardware dependent with an average speedup of x1.4 observed on GeForce™ 8 hardware. A test Direct3D 9 implementation running on a high-end Geforce™ 6 showed similar results. Our results correlate with those obtained by the original authors, as stated in Chan and Durand: "Actual performance gains will depend on the effectiveness of hardware culling. In our tests on NVIDIA hardware, the GeForce 6 (NV40) can rasterize z/stencil (color writes disabled) at 32 pixels/clock, but it can reject pixels at 64 pixels/clock. Similarly, we observed that the GeForce FX 5950 (NV38) can rasterize z/stencil at 8 pixels/clock, but it can reject at 16 pixels/clock. Both measurements represent peak performance. In either case, the relative speedup is a factor of 2".

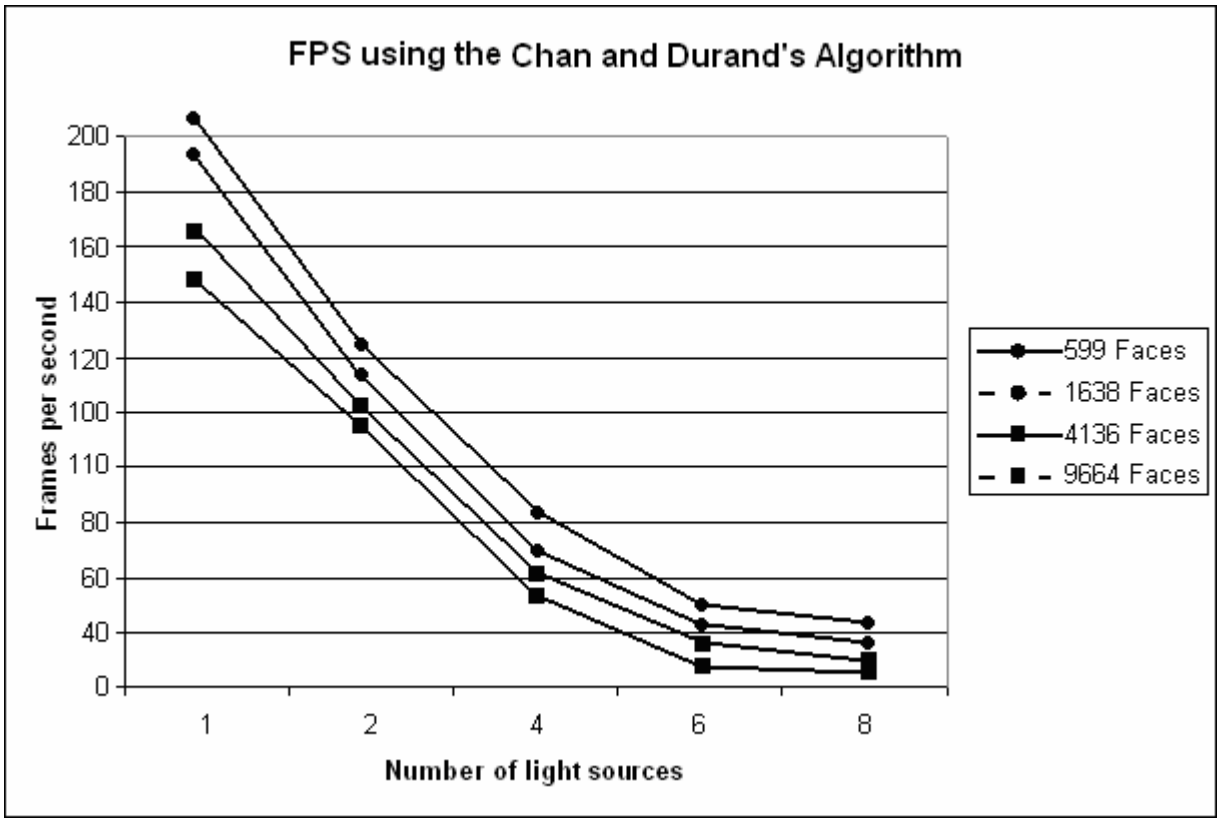


Figure 4.26 The correlation between the number of light sources and FPS rendering performance for polygonal meshes with varying number of faces using Chan and Durand's hybrid approach.

Construction Complexity

Assessing the total number of clock cycles and the number of clock cycles for every routine of Chan and Durand's hybrid shadow construction process allows us to isolate the processor intensive operations and possible areas of improvement. The results shown in Figure 4.27 compare the processor dependence of our 599-face 'tiger' mesh. These tests were executed on the same system as the one used for scalability testing. We also employed an Intel Pentium 4 'Northwood' 2.8 GHz – based test system (with all its hardware components corresponding to that of our AMD test machine). The data obtained from this machine is also shown in Figure 4.27. These results will be used when comparing the standard C/Direct3D implementation to the utilisation of Intel's SSE2 and AMD's 3DNow! instruction sets.

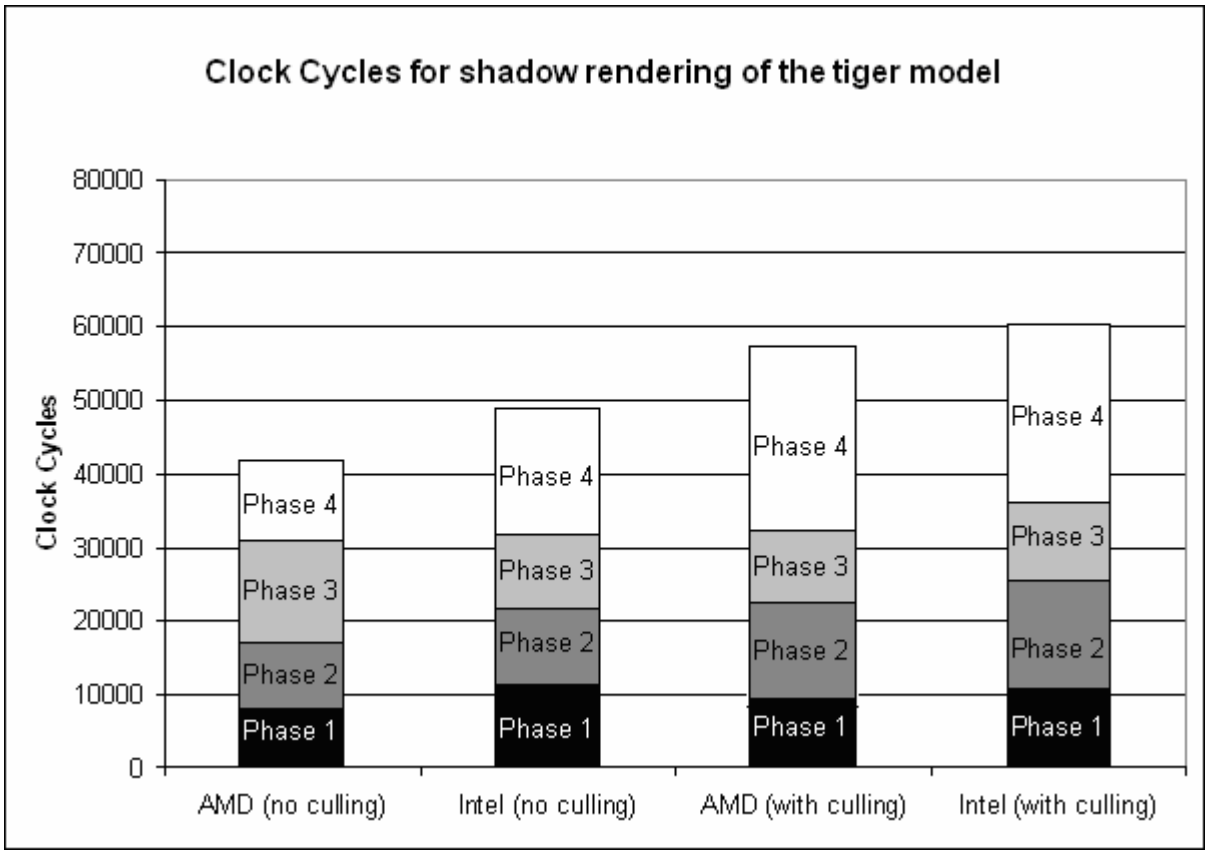


Figure 4.27 Comparing clock cycle iterations with and without culling for the tiger mesh (599 faces) using different hardware configurations.

We investigated the number of clock cycle iterations for each of the shadow volume construction phases. The first phase in this process is to use a shadow map to find pixels in the image that lie near shadow silhouettes. The second phase involves construction of the silhouette polygons (also referred to as the silhouette plane polygons); with the third phase responsible for the construction of the shadow volume's capping triangles. The fourth phase is the actual construction of the shadow volume. We can also modify this phase to construct shadow volumes while at the same time culling polygons located outside of the volume. We can similarly modify the first shadow volume construction phase to calculate the number of light source facing triangles with the culling of polygons outside of the volume. As can be seen in Figure 4.27, the difference between the two hardware configurations is minimal. The culling of polygons located outside of the volume involves more work initially, thus resulting in some performance loss.

We repeated the same experiment on the AMD machine using higher polygon models. As can be seen in Figure 4.28, the total number of clock cycles increases drastically when the number of faces increase, but the relation between the work done during the different phases does not differ significantly. The increasing clock cycle measurements are mostly due to the numerous conditional branches executed during the shadow volume creation and silhouette detection process. Translating and/or rotating a mesh also impacts heavily on the clock cycle count due to the recalculation of the light source facing polygons, the shadow map and subsequently the shadow volume.

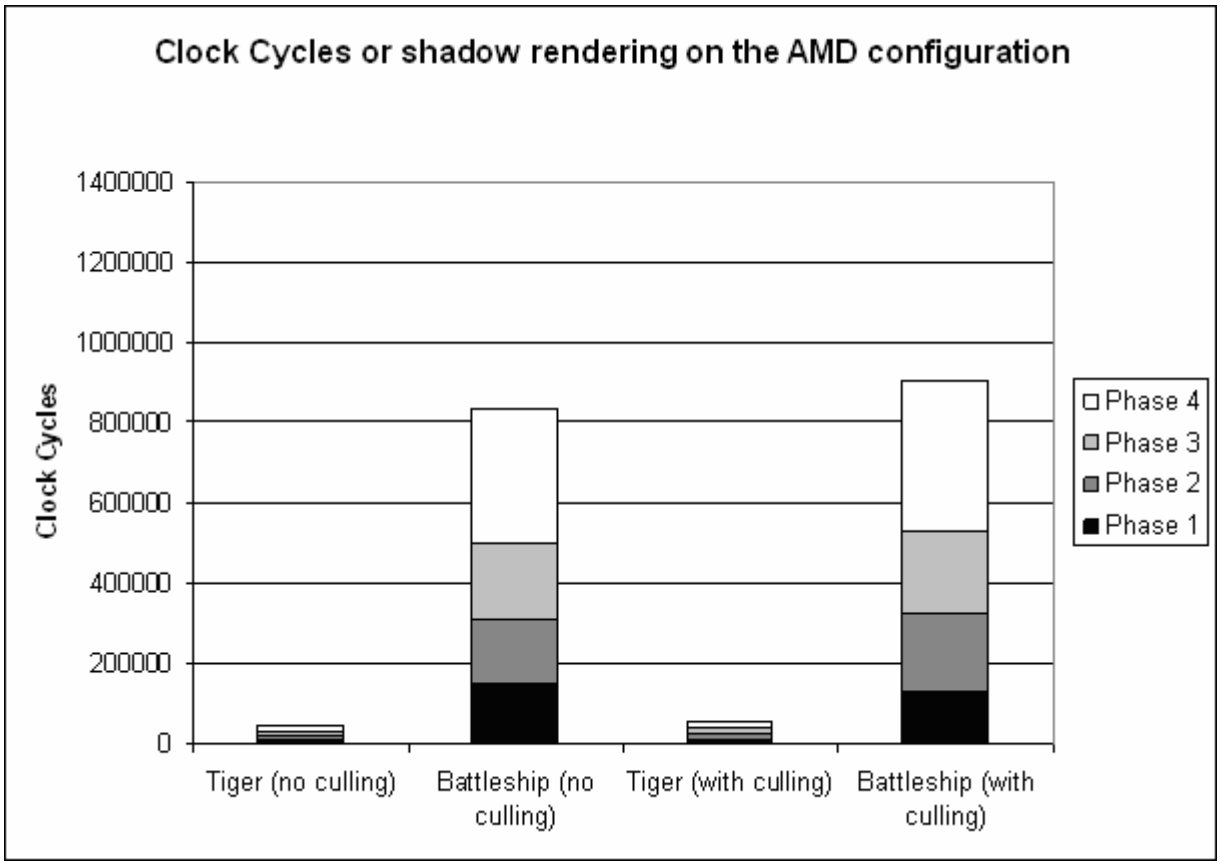


Figure 4.28 Comparing clock cycle iterations for the tiger (599 faces) and the battleship (9664 faces) – shown here with and without culling respectively.

Rendering Accuracy and Detail

The rendering accuracy and detail of projected shadows can be determined by way of visual observation. A model is translated and rotated within a scene and the detail of the projected shadow is investigated. Shadow quality is once again somewhat dependent on the depth map's resolution and shadow edges can, as with McCool's approach, be anti-aliased without any need to recapture the shadow map. Also, the use of a lower resolution shadow map is acceptable due to the shadow volumes being responsible for reconstructing the shadow silhouette. Figure 4.29 (a) shows a properly cropped and skewed shadow rendered using a high resolution depth map with Figure 4.29 (b) showing a portion of the same shadow rendered via a low resolution depth map.

(a) (b)

Figure 4.29 (a) A properly cropped and skewed shadow rendered using a high resolution depth map and (b) showing a portion of the same shadow rendered via a low resolution depth map.

Instruction Set Utilisation

Chan and Durand's hybrid shadow algorithm, as stencil shadow volumes and McCool's approach, requires the CPU for the creation of shadow geometry and for the counting of facing triangles. We will thus observe nearly identical performance gains when utilising Intel's SSE2 or AMD's 3DNow! instruction set (as opposed to conventional sequentially executed routines). There is once again a significant gain, especially in the first phase. It is also noticeable that the second phase did not lend itself to any gain. The difference between the utilised hardware offerings is negligible.

The previously discussed loop counting the number of facing polygons can be subdivided and parallelised via either SSE or 3DNow!, with each individual loop processing a portion of the triangle faces. This parallelised face counting results in a performance increase from 9556 clock cycles to 83 clock cycles for the SSE2 implementation and 8051 to 101 clock cycles for the 3DNow! implementation.

All the other shadow volume construction phases use similar loops with either the culling or the filling of triangle indices added. Partitioning and parallelising these loops via SSE2 or 3DNow! lead to substantial performance increases. The slight architectural differences between the Intel and AMD instruction sets lead to slightly varying results. The AMD architecture barely outperforms the Intel architecture when surfaces outside of the light volume aren't culled. One reason for this could be the AMD processor's clock speed - the Athlon64 3000+ operates at a substantially lower frequency than its Intel counterpart (2.8GHz). It is, however; patently clear that usage of either SSE or 3DNow! results in considerable performance gains.

4.3.5 Elimination of various Shadow Volume Testing Phases (Thakur et al).

To gather the necessary results, we implemented Thakur et al's (2003) algorithm for a number of scenes. In each case, the scene was a relatively simple cubic environment featuring a single movable 3D model and a variable number of light sources. The 3D models utilised are those provided as samples by the Microsoft DirectX SDK. The test system had the following configuration:

NVIDIA GeForce™8800 GTX 768 MB GDDR3 (Video Card),
AMD Athlon™3000+ (Processor),
2.0GB (Memory),
1280x1024 (Screen Resolution).

Scalability

In order to accurately benchmark the scalability of Thakur et al's (2003) algorithm, we start by rendering a somewhat simple scene consisting of a relatively modest polygon model (599 faces) and one light source.

Following this initial rendering, we systematically increase the number of light sources while varying the model complexity. Figure 4.30 summarises the resulting performance data. By analysing these results, it is clear that both the number of light sources and mesh complexity have an influence on the performance of Thakur et al's algorithm. Performance testing of this technique leads to the same conclusions as originally documented by Thakur et al (2003). That is, the method's "storage requirement (like other Shadow Volume methods) grows with the number of objects. However, this method may become more amenable in storage management than its peers. As pointed out earlier, the actual shadow is a logical OR of shadows from all objects. Using this fact greatly reduces the number of lookup calls and so the processing time. Similarly, this concept can also be applied to storage." This technique, to summarise, shows some performance gains over traditional stencil shadow volumes when rendering large geometrically complex scenes with a great number of light sources while allowing for a number of trade offs between speed and quality.

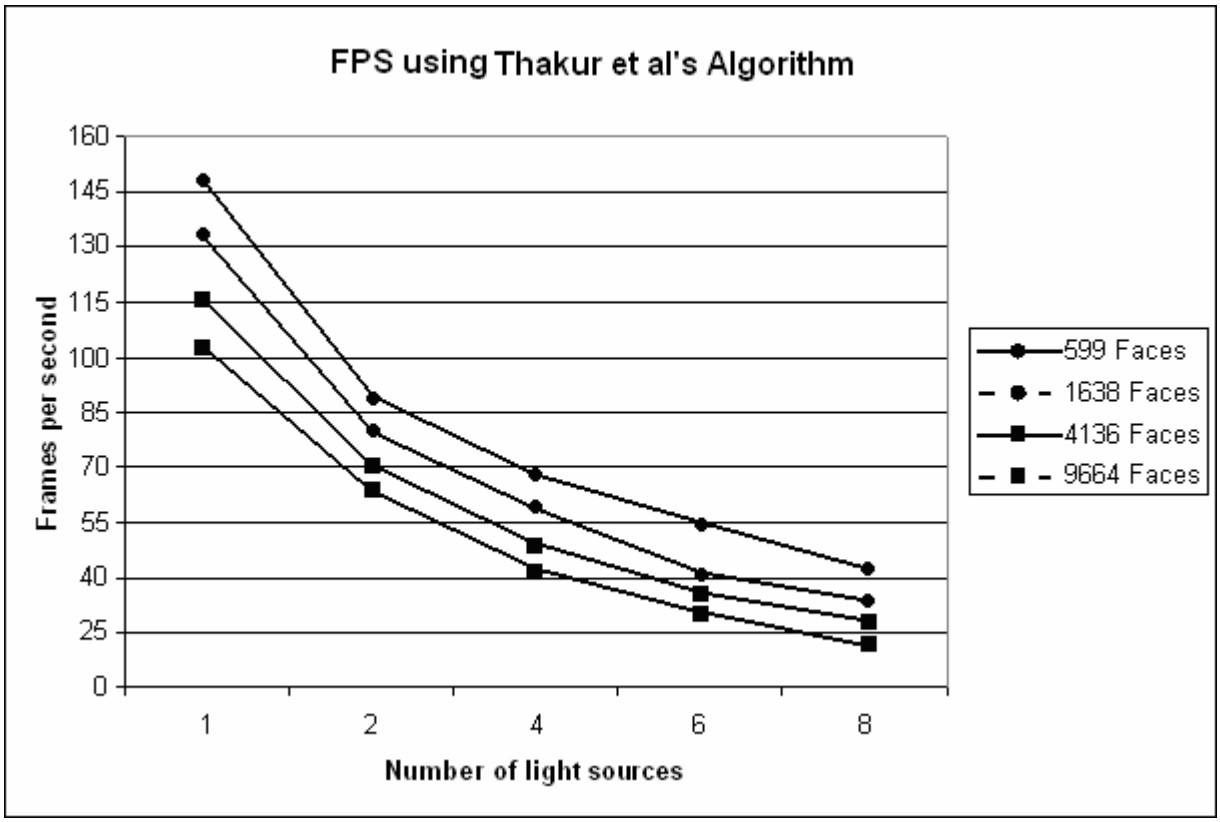


Figure 4.30 The correlation between the number of light sources and FPS rendering performance for polygonal meshes with varying number of faces using Thakur et al's algorithm.

Construction Complexity

Thakur et al's algorithm constructs ridge edge loops (an inexact form of silhouette edges) to determine the shadow volume (without any expensive computations). The silhouette detection process is thus "automatic" in a sense. There is also no need to perform any edge intersection tests or shadow-polygon intersection tests as shadows are discretised into angular spans (corresponding to scan-lines which are in turn stored in a lookup table – data used to directly mark shadowed pixels).

Assessing the total number of clock cycles and the number of clock cycles for every routine of the Thakur et al ridge edge loop construction process allows us to isolate the processor intensive operations and possible areas of improvement. The results shown in Figure 4.31 compare the processor dependence of our 599-face 'tiger' mesh. These tests were executed on the same system as the one used for scalability testing. We also employed an Intel Pentium 4 'Northwood' 2.8 GHz – based test system (with all its hardware components corresponding to that of our AMD test machine). The data obtained from this machine is also shown in Figure 4.31.

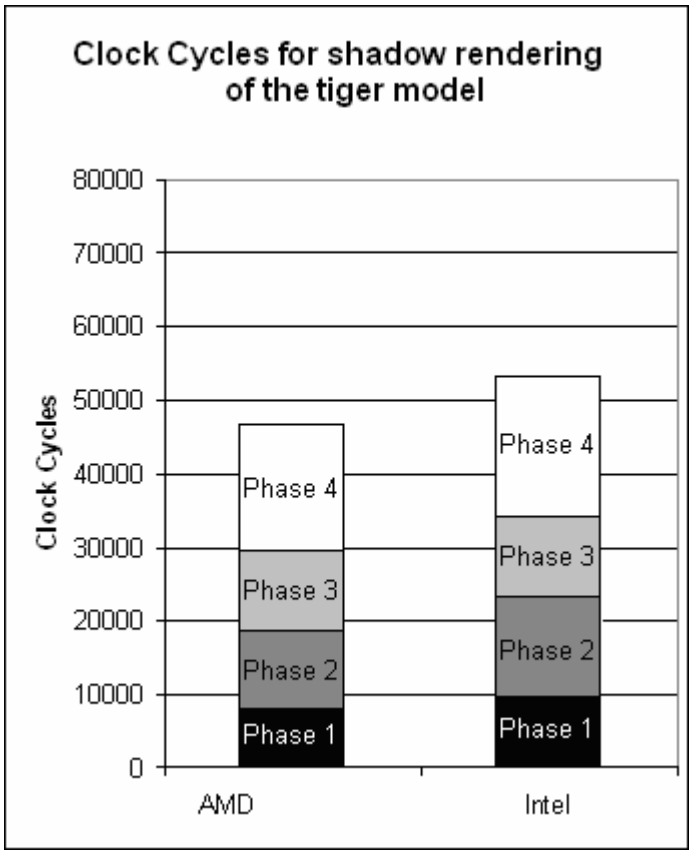


Figure 4.31 Comparing clock cycle iterations of the tiger mesh (599 faces) using different hardware configurations.

We investigated the number of clock cycle iterations for each of the ridge edge loop construction phases. The first phase in this process is to find the ridge edges. The second phase represents the connection of ridge edges to form loops; with the third phase dealing with the calculation of angular coordinates (of vertices positioned on ridge edge loops). The fourth phase is the process where vertices with local peaks in θ (ridge edge loops are sliced along θ with local peaks being specific points in the loop) are identified and where edge points are appended to the lookup table until a minimum in θ is reached (by starting from the identified peaks).

We repeated the same experiment on the AMD machine using higher polygon models. As can be seen in Figure 4.32, the total number of clock cycles increases drastically when the number of faces increase, but the relation between the work done during the different phases does not differ significantly. The increasing clock cycle measurements are mostly due to the numerous conditional branches executed during the ridge edge loop creation process. Translating and/or rotating a mesh also impacts heavily on the clock cycle count due to the recalculation of the light source facing polygons and subsequently the shadow volume.

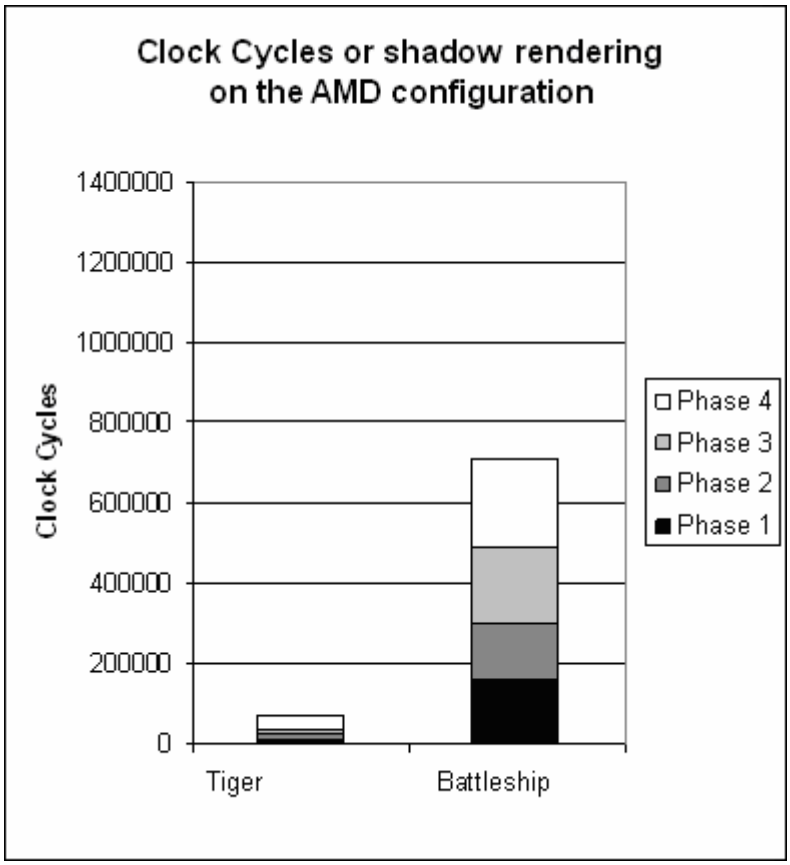
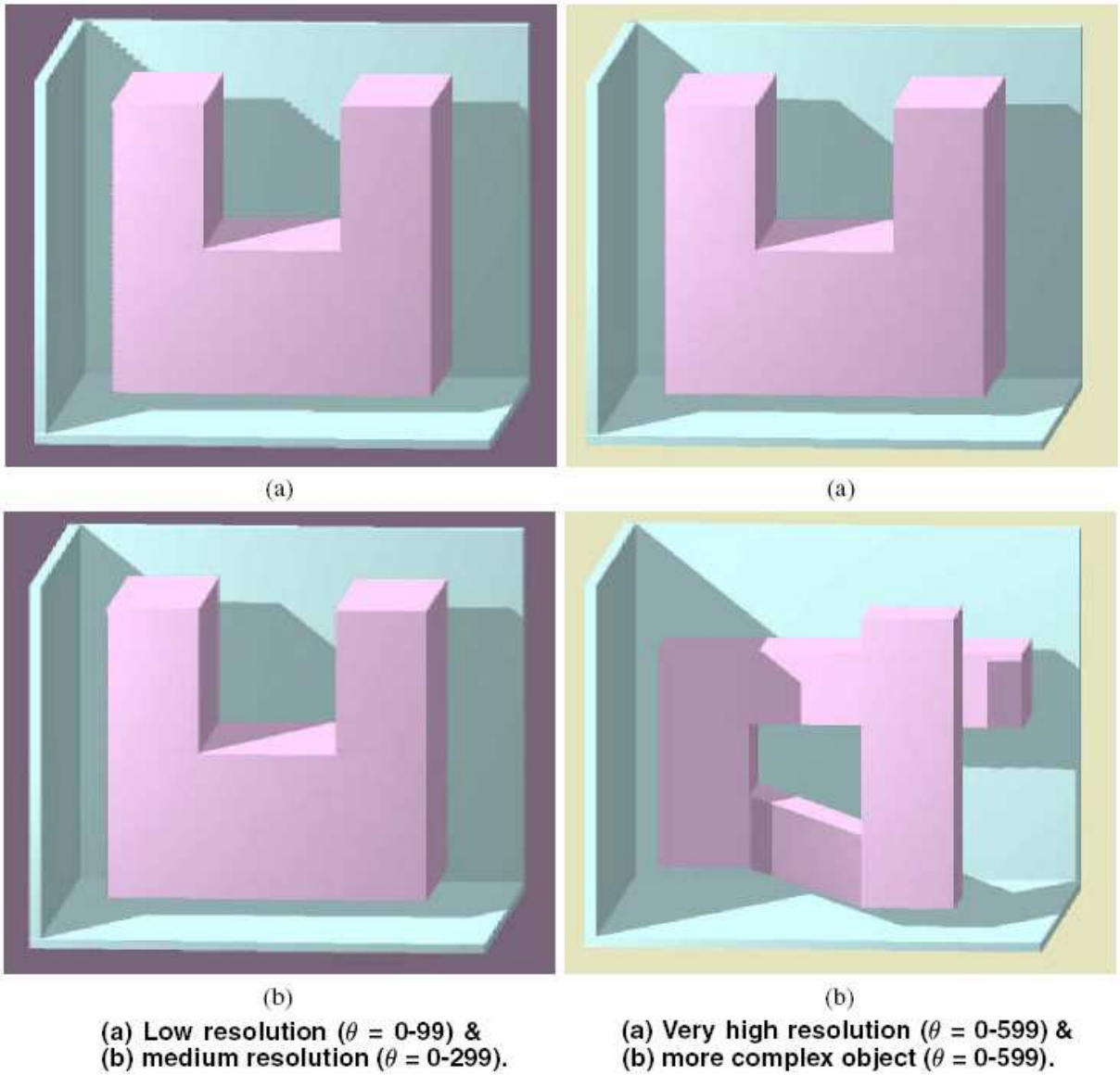


Figure 4.32 Comparing clock cycle iterations of the tiger (599 faces) and battleship (9664 faces) mesh respectively.

Rendering Accuracy and Detail

The rendering accuracy and detail of projected shadows can be determined by way of visual observation. A model is translated and rotated within a scene and the detail of the projected shadow is investigated. Shadow quality is heavily dependent on $\theta's$ resolution, however, increasing the resolution beyond the native screen resolution has no effect – the screen resolution poses an upper limit. Figure 4.33 shows a properly cropped and skewed shadow with Figure 4.34 (taken from Thakur et al, 2003) illustrating the effect of $\theta's$ resolution on image quality.

Figure 4.33 A properly cropped and skewed shadow.



(a) Low resolution (θ = 0-99) & (b) medium resolution (θ = 0-299).

(a) Very high resolution (θ = 0-599) & (b) more complex object (θ = 0-599).

Figure 4.34 The effect of $\theta's$ resolution on image quality.

Instruction Set Utilisation

Thakur et al's algorithm utilises the CPU to construct not only the lookup table (see section 3.3.3) but also to perform scan-line conversion and generate a query at each point (x, y, z) to determine whether the point's in shadow or not (and to perform the subsequent shadow query). We will thus observe some performance gains when utilising Intel's SSE2 or AMD's 3DNow! instruction set (as opposed to conventional sequentially executed routines).

4.3.6 Shadow Volumes and Spatial Subdivision

We now present the results obtained from combining the depth-fail stencil shadow volume algorithm with spatial subdivision. This unification results, as previously mentioned, in real-time frame rates for rather complex scenes.

Our approach, as discussed, uses the Octree to calculate the shadow volume unification by traversing the tree in a front-to-back order, thus in effect subdividing the surface (endpoint) polygons for each element/object. This arrangement results in performance gains where the rendered scene contains several concealed shadow volumes. However, the Octree structure does require some processing time to build and the non-Octree enhanced shadow volume algorithm performs much better where there are few concealed shadow volumes or in situations where light sources are dynamically added or removed.

To gather the necessary results, we implemented our algorithm for a number of scenes. In each case, the scene was a relatively simple cubic environment featuring a single movable 3D model and a variable number of light sources. The 3D models utilised are those provided as samples by the Microsoft DirectX SDK. The test system had the following configuration:

NVIDIA GeForce™8800 GTX 768 MB GDDR3 (Video Card),
AMD Athlon™3000+ (Processor),
2.0GB (Memory),
1280x1024 (Screen Resolution).

Scalability

In order to accurately benchmark the scalability of our algorithm, we render a somewhat simple scene consisting of a relatively modest polygon model (599 faces) and one light source.

Following this initial rendering, we systematically increase the number of light sources while varying the model complexity. Figure 4.35 shows the frame rates obtained when using our technique with a varying number of light sources and

models of differing complexity. Similar to the depth-fail stencil shadow volume algorithm, both the number of faces and number of light sources have a negative impact on the overall frame rate.

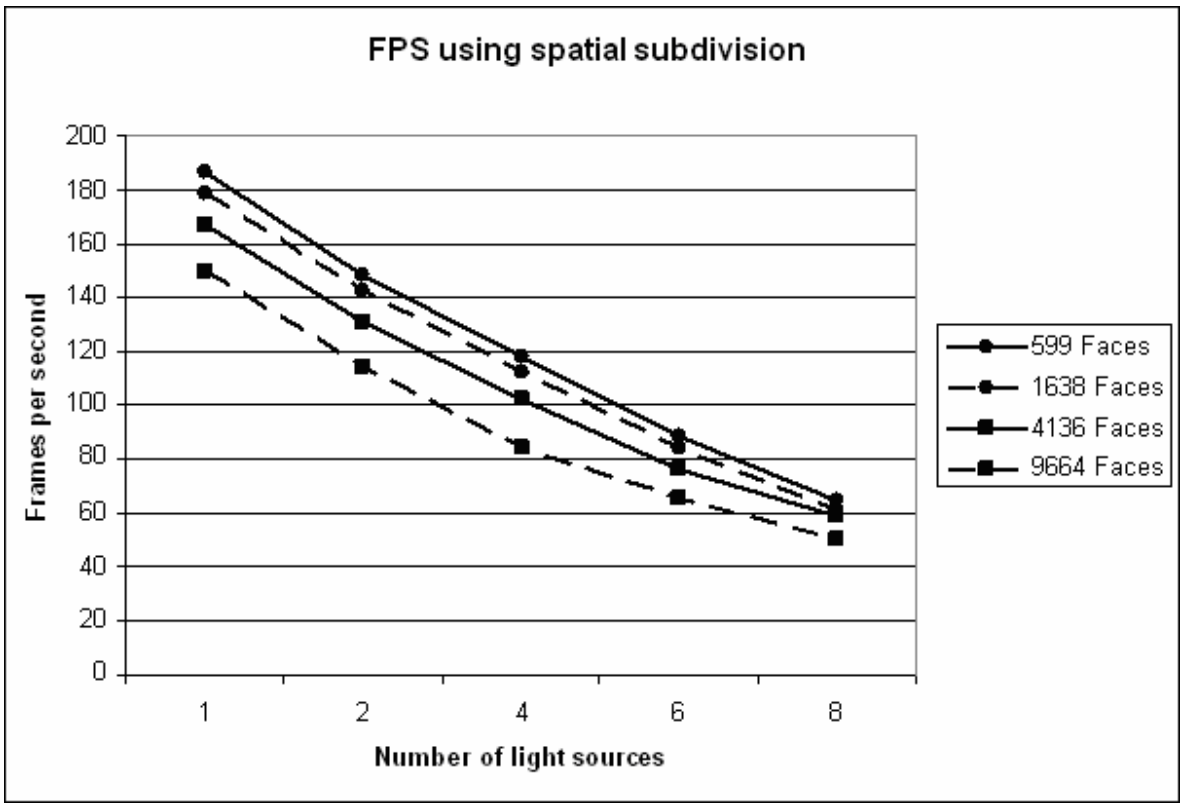


Figure 4.35 The correlation between the number of light sources and FPS rendering performance for polygonal meshes with varying number of faces using our approach.

Construction Complexity and Instruction Set Utilisation

Our shadow algorithm, as stencil shadow volumes and McCool's approach, requires the CPU for the creation of shadow geometry and for the counting of facing triangles. It also utilises CPU power to populate the octree data structure. The next step is thus to extend this spatial subdivision approach by combining it with the utilisation of the SSE2 instruction set (the data obtained from the use of 3DNow! is nearly identical).

Figure 4.36 shows the results obtained from this hybrid technique. The improvement is in the order of about 10%.

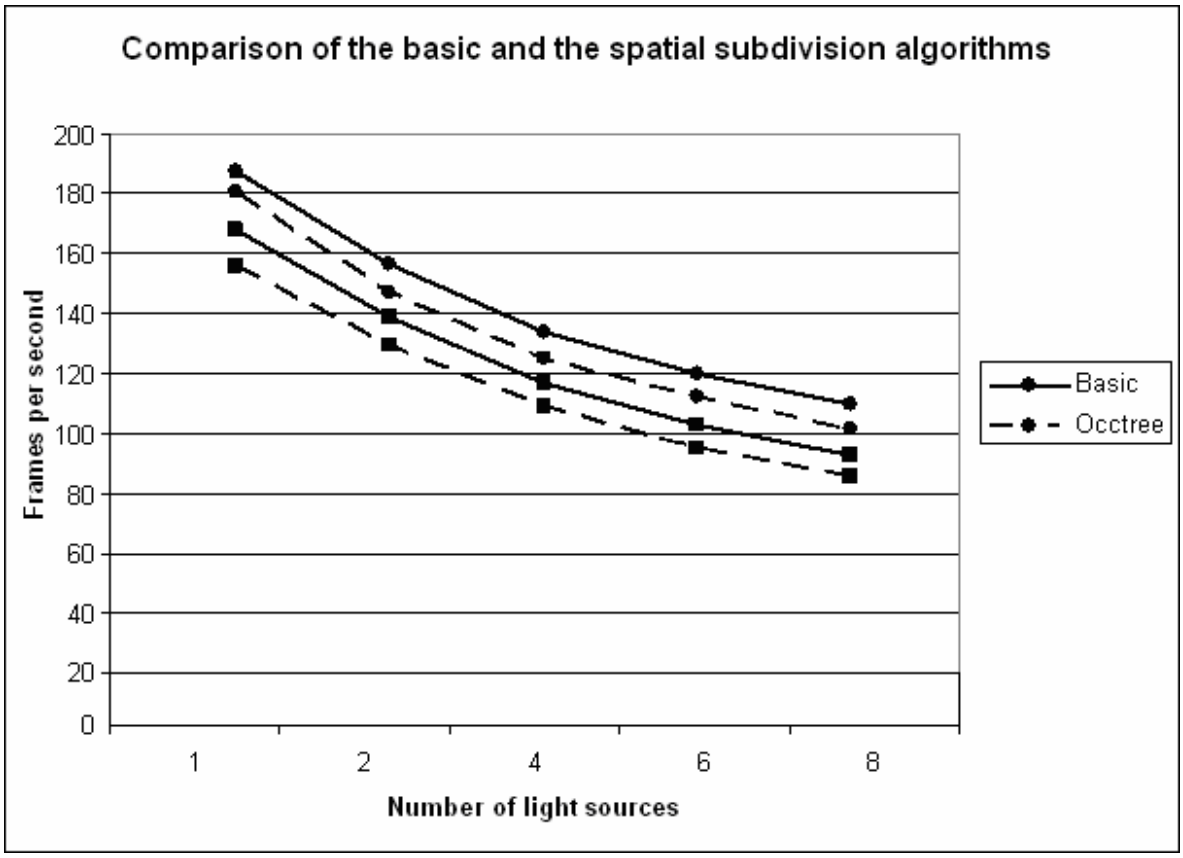


Figure 4.36 The correlation between the number of light sources and FPS rendering performance for polygonal meshes with varying number of faces using our extended approach.

Rendering Accuracy and Detail

The rendering accuracy and detail of projected shadows can be determined by way of visual observation. A model was translated and rotated within a scene and the detail of the projected shadow was investigated. Since the algorithm only adds an underlying data structure to speed up calculation, and does not alter any of the techniques applied by the basic stencil shadow volume algorithm to determine the shadowed pixels, the shadow quality is identical to that of the original stencil shadow volume algorithm (shown in Figure 4.37).

Figure 4.37 A properly cropped and skewed shadow.

4.4 Succinct Algorithm Comparison

This section presents a comparison between the results documented in section 4.3 (see Figure 4.40 for the mean performance of each algorithm). It also summarises these results with specific emphasis on the most appropriate application areas.

It is important to note that our spatial subdivision algorithm was analysed in a statically lit environment. This results in its relatively high performance when compared to the other algorithms. Its performance was found to be comparable to the basic stencil shadow volume algorithm in situations where dynamic lighting is implemented. Our algorithm will thus outperform all other algorithms where light sources aren't added, moved or removed.

In Figure 4.38 the frame rates achieved (for the 1638 faces model) via the implementation of spatial subdivision is compared to that obtained using the Heidmann algorithm. It is clear from the data that our approach results in significantly better performance than the original stencil shadow volume algorithm (40% better for one light source and 200% better for eight).

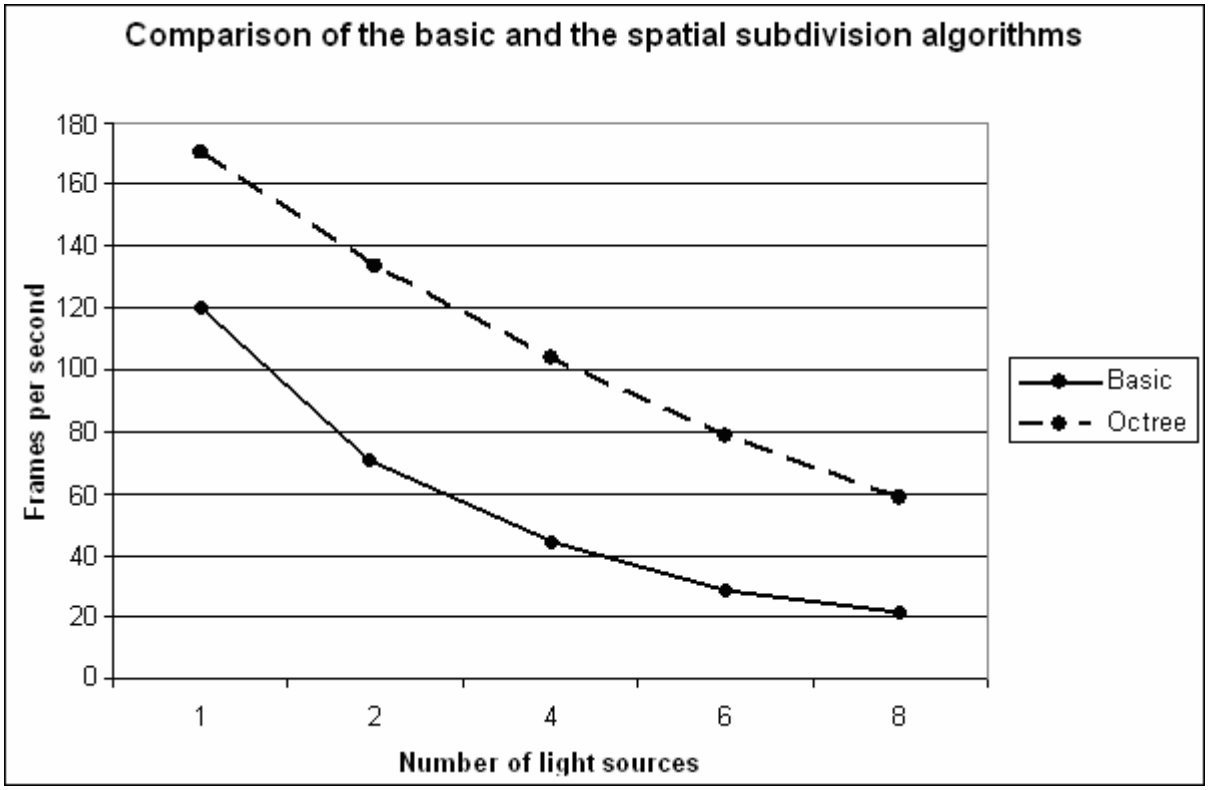


Figure 4.38 Comparison of our approach with the depth-fail stencil shadow volume approach.

In Figure 4.39, the frame rates attained from using spatial subdivision combined with the utilisation of the SSE2 instruction set is compared to that obtained from using the Heidmann algorithm.

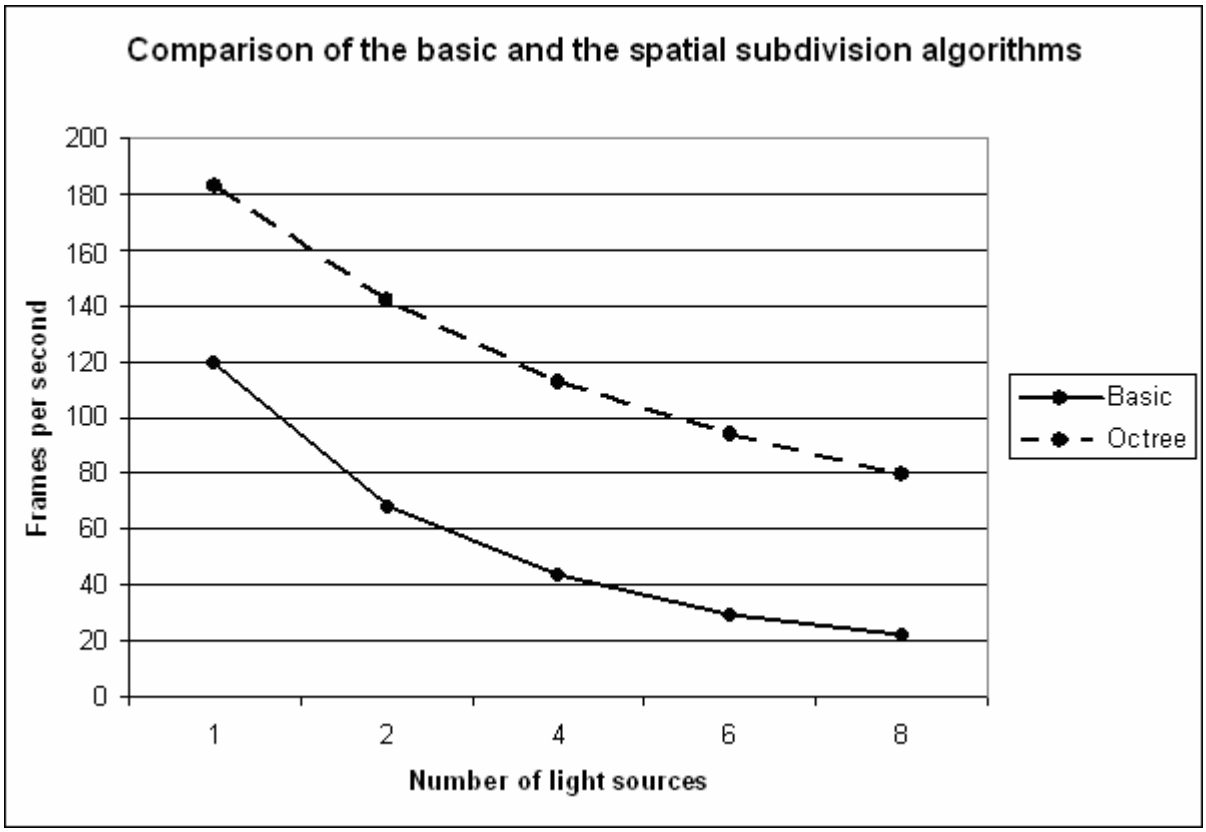


Figure 4.39 Comparison of our extended approach with the depth-fail stencil shadow volume approach.

From the results given in Figure 4.40, it is clear that our spatial subdivision approach combined with hardware extensions offers the best performance for statically lit scenes. This algorithm does, however, require processing time for Octree-construction and the non-Octree enhanced shadow volume algorithms perform much better in situations where light sources are dynamically added, moved or removed. We will use the spatial subdivision approach coupled with SSE2/3DNow! utilisation for all environmental areas lit using static light sources.

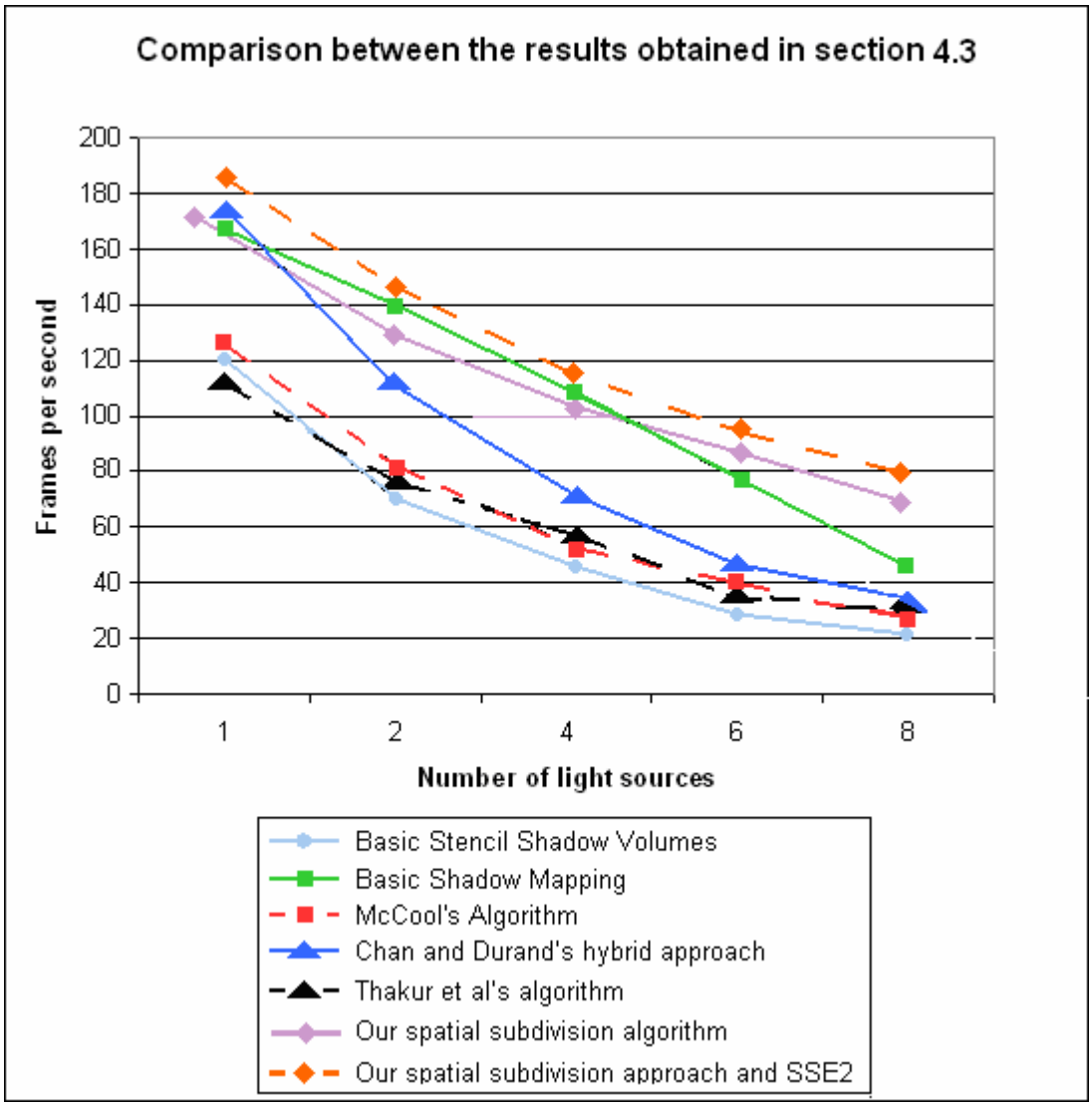


Figure 4.40 Comparison of all the previously listed algorithms (1-8 light sources).

Chan and Durand's (2004) algorithm is the second best algorithm when rendering high-quality shadows with only a single dedicated light source. This algorithm shows significant performance degradation when more light sources are added. It does, however, outperform all the remaining shadow volume-based algorithms for up to eight light sources. We will use Chan and Durand's algorithm for all scenes consisting of eight or less dynamic light sources when high-quality shadows are required.

The shadow mapping algorithm performs only slightly worse than Chan and Durand's (2004) algorithm (when rendering scenes consisting of just one light source). That said, our critical analysis implementation does render low-resolution shadow maps. However, increasing this shadow map resolution will have a net-negative impact on our overall rendering performance. We will use shadow mapping (with average shadow resolution) for all scenes consisting of two or more dynamic light sources and where the shadow casting objects are located a significant distance from the point-of-view.

McCool's (2000) algorithm is the second best choice when dealing with scenes featuring one to eight light sources and when high-quality shadows are required. We won't be utilising this algorithm, rather opting for Chan and Durand's hybrid approach. The same goes for the classic stencil shadow volume algorithm and Thakur et al's (2003) algorithm.

The previous comparison (given in Figure 4.40) only deals with a limited number of light sources. The choice between the most appropriate algorithms is, however, mostly superficial due to 200 frames per second and 60 frames per second displaying similar to the human eye. It is only when frame rates fall below 30 per second that we start to notice. Running the same simulations (but with the light source count ranging from nine to sixteen) shows a rapid decrease in our frames per second performance. Figure 4.41 shows these results.

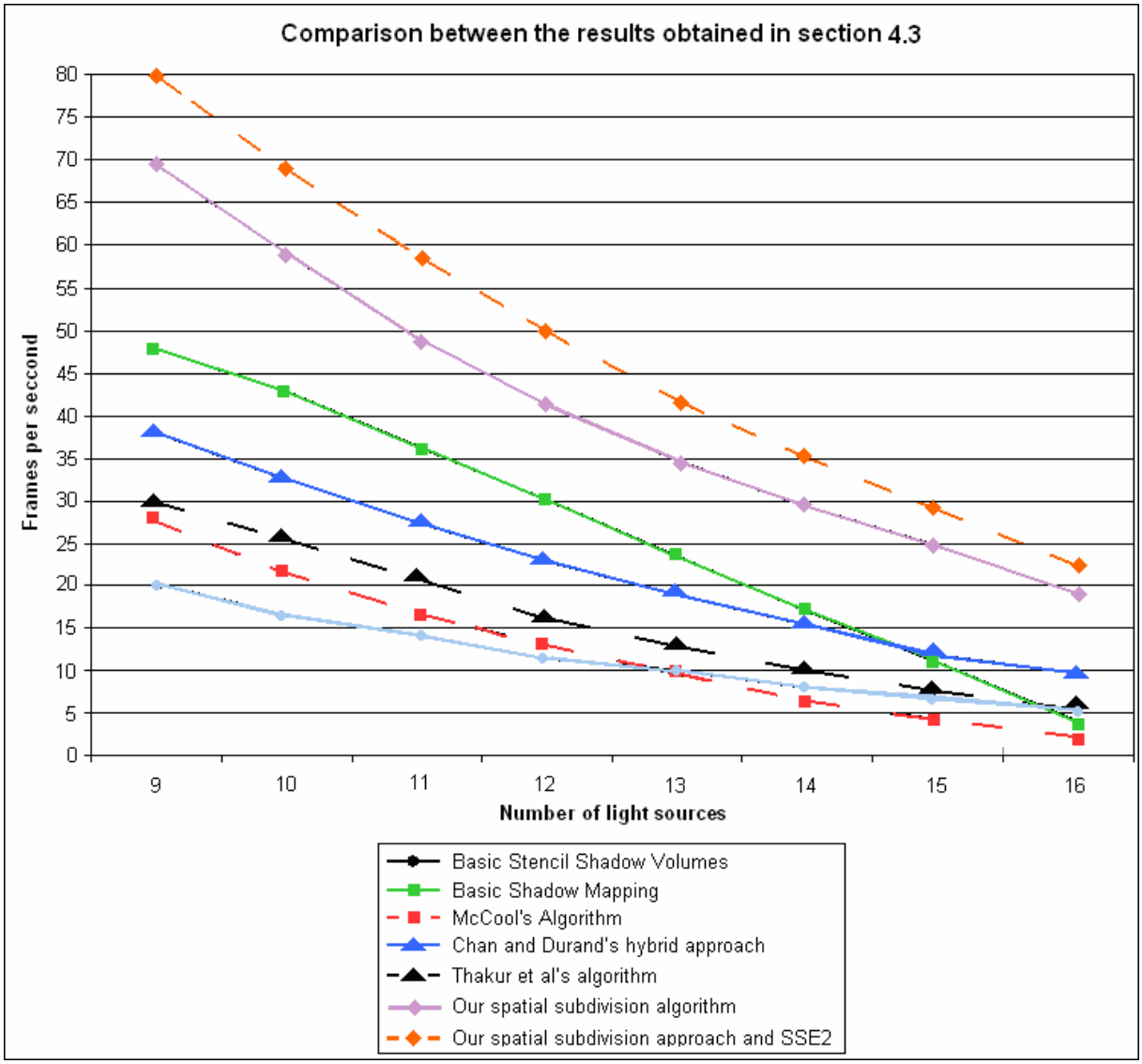


Figure 4.41 Comparison of all the previously listed algorithms (9-16 light sources) – please note; our spatial subdivision algorithm was analysed in a statically lit environment, thus resulting in its high performance (its performance is comparable to the basic stencil shadow volume algorithm in situations where dynamic lighting is implemented).

Considering Figure 4.41, we can once again see our spatial subdivision approach coupled with the SSE2/3DNow! instruction set outperforming all the other algorithms. This algorithm is, as mentioned, only amenable to environments utilising static lighting – making the comparison a bit bias. We will, however, continue to use this algorithm for all environmental areas lit using static light sources.

The Basic shadow mapping algorithm remains the best choice when dealing with dynamically lit environments, at least when working with fourteen or less light sources and when shadow rendering quality isn't as important (see Figure 4.40 and 4.41). We will continue to use shadow mapping for all scenes consisting of more than two and less than fourteen dynamic light sources and where the shadow casting objects are located a significant distance from the point-of-view. Chan and Durant's algorithm will, however, prove a better choice for both close range and distant objects when rendering scenes consisting of fourteen or more dynamic light sources. We will also use Chan and Durrand's algorithm for all scenes consisting of nine or more dynamic light sources when high-quality shadows are required.

4.5 Summary

In the chapter we started by presenting our benchmarking mechanism and a set of criteria for the evaluation of shadow generation techniques. The given evaluation criteria were selected with the aim of assessing the relationship between shadow rendering quality and performance – in turn allowing us to isolate key algorithmic weaknesses and possible bottleneck areas.

Specific algorithms benchmarked and analysed include: the basic stencil shadow volume algorithm, the basic hardware shadow mapping algorithm, McCool's shadow volume reconstruction using depth maps, Chan and Durand's hybrid algorithm for the efficient rendering of hard-edged shadows, Thakur el al's algorithm based on the elimination of various shadow volume testing phases and our own algorithm based on shadow volumes, spatial subdivision and instruction set utilisation. We found the spatial subdivision approach combined with hardware extensions to offer the best performance when rendering statically lit scenes. We subsequently identified Chan and Durand's algorithm to be the best algorithm when rendering scenes consisting of eight or less dynamic light sources when high-quality shadows are required and where shadow casting objects are located near the point-of-view. Shadow mapping was, in turn, identified as the best approach when rendering scenes consisting of more than two and less than fourteen dynamic light sources and where the shadow casting objects are located a significant distance from the point-of-view. Chan and Durant's algorithm is, however, a better choice for both close range and distant objects when rendering scenes consisting of fourteen or more dynamic light sources. We will also use Chan and Durrand's algorithm for all scenes consisting of nine or more dynamic light sources when high-quality shadows are required. McCool's algorithm is the second best choice when dealing with scenes featuring one to eight light sources and when high-quality shadows are required. We will not be utilising this

algorithm, rather opting for Chan and Durand's hybrid approach. The same goes for the classic stencil shadow volume algorithm and Thakur et al's algorithm.

The chapter concluded with a comparison between the results obtained; we also summarised these results with specific emphasis on the most appropriate application area. The next chapter provides an overview of expert systems and fuzzy-logic – two elements incremental to the implementation of our high speed shadow rendering framework.

Chapter 5

Expert Systems and Fuzzy Logic

This chapter provides a brief overview of a number of basic concepts pertaining to expert systems and fuzzy logic. This overview is included to illuminate the utilisation of these concepts with regards to designing and implementing a system for high-speed shadow rendering (discussed in Chapter 6). Detailed discussions of these topics can be found in a large number of textbooks such Nilsson (1986) and Coppin (2004) to name a few.

Chapter 5 focuses on the theoretical aspects of expert systems, their architecture and a number of advantages and disadvantages inherent to their use (as far as it applies to the design of our system). Following this discussion, it deals with the concept of fuzzy-logic and its use in the dynamic selection of shadow rendering algorithms.

It is not the intention of this dissertation to contribute to the domain knowledge of artificial intelligence. It is merely applying well established techniques to the terrain of real time shadow rendering. It was decided to include this textbook knowledge for the sake of the reader's convenience and may be skipped by individuals who are familiar with these concepts.

In this chapter we will investigate:

- The architecture of a basic rule-based expert system
- Rule-based decision making
- Fuzzy reasoning and the concept of fuzzy expert systems
- Linguistic variables
- Fuzzy sets
- Membership functions
- Mamdani inference

5.1 Introduction

A technique often employed to facilitate the storage of and access to human expertise, accumulated through training and experience, is that of *expert systems* (also called knowledge systems). An expert system is based on three concepts, namely, the knowledge of facts, data about the relationship among these facts and a method to store and access this data (Ignizio, 1991:127). An expert system is created by extracting facts from human knowledge such as routines, historic events, relationships, etc and transforming this data so that it can be used to influence the reasoning of a computer controlled opponent playing against the user, for example (Giarratano et al, 2005 :28).

The most basic expert system is defined by a set of production rules in turn used to analyse information (Salton, 1987). This mathematical analysis yields a recommendation with regards to the course of action that should be taken by the user, a subsystem or algorithm.

Expert systems can, for example, be used in games to implement the reasoning of computer controlled opponents. When used in such a way, the knowledge system is defined to extract and store facts about the human players. Production rules (simple if-conditionals) can subsequently be used to reason and take certain actions based on processed human knowledge.

An expert system, as mentioned, analyses a dataset to determine the best solution to a given problem. Our real-time shadow generation framework employs such a system. This system specifically consists of an empirically ascertained dataset and a collection of rules to analyse the data and information of various elements pertaining to the scene currently being rendered.

We will now investigate the theoretical aspects of expert systems, their architecture and a number of advantages and disadvantages inherent to their use. Following this discussion, we will look at the extension of expert systems to include fuzzy logic based reasoning.

5.2 Expert Systems

The knowledge stored in an expert system is obtained via training, empirical experimentation and prior human experience. Raw real-world knowledge is subsequently transformed into an expert system during a process called *knowledge engineering*. This process or task, performed by a knowledge engineer, encapsulates the construction of an expert system. The finalised expert system consists of the following elements: a set of facts, the relationship among these facts and a mechanism for the storage and rapid retrieval of this information (Ignizio, 1991:59).

Expert systems are commonly constructed via a series of *production rules* (common if-then-else structures). These production rules are used to select an appropriate action based on the truthfulness of a certain condition; for example, in the most basic terms: if some condition is true, take some action. A condition is known as the *antecedent* with the action called the *consequent*. The antecedent expresses some fact stored in the expert system's knowledge base with the consequent triggering some event. This event can either be an action to perform or the addition or removal of a stored fact.

Expert systems differ from simple conditional programming with regard to the order of execution. For example, a common sequence of if-else statements, as found in the C programming language, is executed in a predefined order. Expert systems make use of an *inference engine* to select conditionals based on current real-time data. The action taken is only governed by the current facts and rules, not by any predefined execution order.

Expert systems are frequently encountered in human resources departments, the medical world, banking and process control environments. For example, a human resources department might use such a system to calculate an individual's salary bracket or a medical centre might employ an expert system to diagnose a certain health condition based on a number of symptoms.

We will use an expert system to control the real-time selection of shadow rendering algorithms. The knowledge base of this expert system consists of experimental results obtained through the critical analysis of numerous real-time shadow rendering algorithms (and the improvements made through the use of various hardware extensions and hybrid approaches). Production rules are implemented via an inference engine. This inference engine is in turn used to select the most appropriate algorithm based on certain properties of the scene being rendered. For instance, our system could contain the following production rule: if there are a lot of light sources in a scene and the scene has a high geometric complexity, then enable a hybrid stencil shadow volume/shadow mapping algorithm. The notions "a lot of light sources" and "high geometric complexity" are not quantitative facts. Fuzzy logic provides a solution to this problem by assigning quantitative values and/or ranges to these concepts. The concepts "a lot of light sources" and "high geometric complexity" can also be combined into the new one "overly complex", resulting in a new production rule. Our framework combines production rules with fuzzy logic to explicitly symbolise data. This is followed by the selection of the most efficient shadow rendering algorithm. It is important to note that this framework is also adaptable for use with other rendering algorithms such as real-time reflections and particle systems.

A basic rule-based expert system consists of the following modules:

- An inference engine or interpreter.
- A fact database.
- A knowledge base.
- A User Interface.
- An Explanation System.

- A Knowledge Base Interface.

The knowledge base is nothing more than a database of rules. These rules symbolise the stored knowledge. The fact database embodies the expert system inputs which are subsequently used to make decisions and/or to take certain actions. The inference engine makes the actual decision by combining these expert system rules and facts. The *explanation system* generates information about the manner in which a decision was made with the *knowledge base interface*. It also gives the user access to it and allows the user to edit information stored in the knowledge base (Salton, 1987). Figure 5.1 illustrates the architecture of a typical expert system.

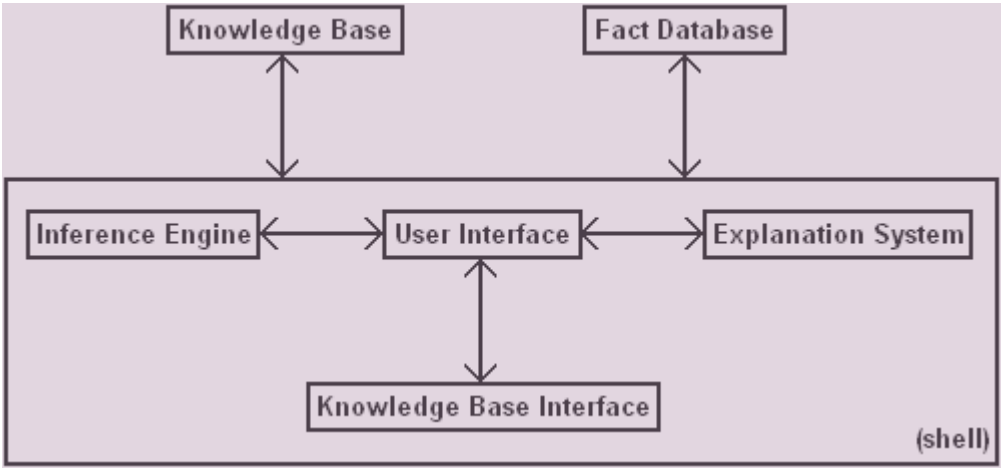


Fig 5.1 Architecture of an expert system.

An expert system's inference engine, explanation system, knowledge base interface and user interface are contained within a "shell". The knowledge base and fact database are connected to this shell in a plugin-like fashion. An *expert system shell* is thus used to define a generic expert system, with the expert system's functionality controlled by the connected fact database and knowledge base. Well-known expert system shells include CLIPS (C Language Integrated Production System), the Java Expert System Shell (a Java implementation of CLIPS), LogicNets (a web-based expert system modelling environment), the SHINE (Spacecraft Health INference Engine) real-time expert system and PyKe (a knowledge based expert system) – Table 5.1 gives a short description of each of these.

Expert System Shell	**Description**
CLIPS	CLIPS is a productive development and delivery expert system tool which provides a complete environment for the construction of rule and/or object based expert systems. Created in 1985, CLIPS is now widely used throughout the government, industry, and academia. (http://clipsrules.sourceforge.net/).
Java Expert System Shell	A CLIPS engine implemented in Java used in the development of expert

	systems (http://www.jessrules.com/).
LogicNets	A web-based expert system modelling environment (http://www.logicnets.com/).
SHINE	A software-development tool for knowledge-based systems and has been created as a product for research and development by the Artificial Intelligence Group, Information Systems Technology Section at NASA/JPL (http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/14039/1/00-0425.pdf).
PyKe	A knowledge-based inference engine/expert system (http://pyke.sourceforge.net/).

Decisions are generally made in one of two ways: forward chaining or backward chaining. *Forward chaining* utilises deduction to reach a result or conclusion. *Backward chaining*, on the other hand, starts at the end-result or conclusion from where it backtracks to determine whether the system has any data which will satisfy this goal or result.

Forward Chaining

An expert system employing a forward chaining strategy will start from a set of facts and, based on a number of rules, attempt to make an accurate decision. The following outlines the steps of such an expert system:

1) Read the expert system inputs from the fact database.
2) Compare the read inputs to the rules in the rule database.
3) If an input fact matches all the antecedents of a rule:
    a. Trigger the rule and add its conclusion to the fact database.
        i. If the conclusion is an action:
            - Execute the action.

For example, consider the following set of rules that is used to select a shadow algorithm based on the number of light sources in a scene:

Rule #1
If there are more than twenty light sources,
Then render all shadows via the depth-pass stencil shadow volume algorithm.

Rule #2
If there are less than twenty light sources,
Then render all shadows via the hardware-accelerated shadow mapping algorithm.

Say we have loaded the following fact into our expert system:

Fact
The scene contains 39 light sources.

The expert system can now examine the rule database, upon which the loaded fact will be matched with rule #1. Hence, rule #1 fires with its result being selection of the depth-pass stencil shadow volume algorithm.

Our expert system implementation utilises a forward chaining strategy to determine results from a collection of rules and facts. Chapter 4 discusses this implementation in much more detail.

Backward Chaining

Forward chaining does, in some cases, suffer from inefficiency due to a number of irrelevant conclusions being reached or more than one rule for a given fact being triggered. *Conflict resolution* is often needed to determine the rule that should be fired. The simplest way of dealing with conflicts is to assign each rule a priority level (Ignizio, 1991:85). The rule with the highest priority is thus fired whenever a conflict occurs. Another solution to this problem is backward chaining. Backward chaining, as mentioned, starts from the end-result or conclusion. From there it backtracks to determine how the given goal or result can be reached given the rules and facts in the system. The following outlines the steps of an expert system utilising such a backward chaining strategy:

1) Start at the goal state – the desired condition.
2) Examine the goal state and analyse the expert system's rules and facts to identify the action(s) that might lead to this goal.
3) Terminate when the start state is reached – at which point a plan would be formulated.

Our expert system implementation operates on a readily available set of facts without the need to prove the validity of specific conclusions. A backward chaining strategy is thus not appropriate in our situation.

There are several advantages inherent to the use of expert systems, specifically; generated results are always consistent, regardless of variations in the input data, a large volume of data can be stored and processed in real-time and decision making occurs in a logical fashion. That said, expert systems will never demonstrate the same "common sense" as human beings with regard to decision making or problem solving. The knowledge base might also contain errors which will, in turn, result in flawed reasoning and skewed results.

5.3 Fuzzy Reasoning and Fuzzy Expert Systems

Fuzzy logic, introduced in the 1960s by LA Zadeh, is an extension of traditional predicate/Boolean logic. Boolean logic, bivalent in nature (true or false), was broadened by this superset to include half-truths (values between "absolutely true" and "absolutely false") (Zadeh, 1965).

Fuzzy logic reasons about so-called fuzzy sets (Chen, 1996). *Fuzzy sets*, introduced by LA Zadeh in 1965, extends traditional sets (where an element can either be present or absent from a set) to define a set based on partial membership (Zadeh, 1965). An element can thus be a member of a set to some degree while at the same time not being a member of the set to some degree. For example, consider a set of colours, `C = {Yellow, Orange, Red, Purple}`, and the colour elements `Burgundy` and `Gold`. `Burgundy` is of course a shade of red with a touch of purple with `Gold` being a yellowish orange colour. `Burgundy` and `Gold` are thus part of set `C` to some degree but they are also non-members of the set to some degree – `Gold` can be considered "fairly yellow" with `Burgundy` being "fairly red". Fuzzy logic is thus based on the extent of truth of a statement with regard to its degree of membership to the set. Figure 5.2 shows the degree of membership of the colour `Gold` with regards to the set element `Yellow`.

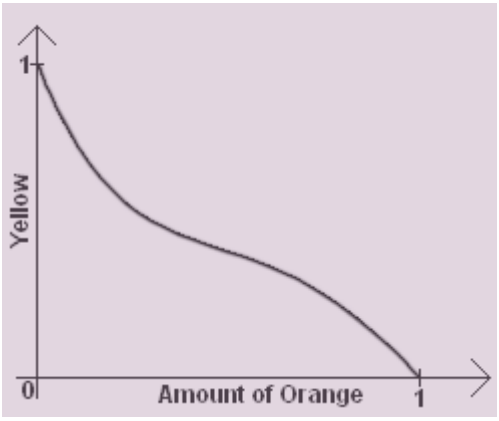


Fig 5.2 The degree of membership of the colour `Gold` (in terms of the amount of orange present) with regards to the set element `Yellow`.

A membership function is used to define a fuzzy set. Such a function assigns a value from an interval to a criteria/element collection (Zadeh, 1965). For instance, the membership function for a fuzzy set C can be written as follows:

$$M_C(x) = \begin{cases} x+9 \; for \; x \leq 15 \\ 15 \; for \; x > 15 \end{cases}$$

A fuzzy logic based expert system utilises a set of linguistic variables (related to the problem) and several membership functions. Fuzzy rules are derived from these variables as well as the knowledge base. These rules are applied by means of Mamdani fuzzy inference. *Mamdani inference*, as proposed in 1975 by Ebrahim

Mamdani, applies a set of fuzzy rules on a set of traditional precise inputs to obtain a precise output value (such as an action recommendation) (Mamdani et al, 1975).

To understand Mamdani inference, consider the following example (please note, this example does not represent the rules that were actually used, the given sets and functions are simply examples). Say we have the following rules:

Rule #1
If there are few light sources,
Then render all shadows via the hardware-accelerated shadow mapping algorithm.

Rule #2
If there are many light sources,
And the polygon complexity of the scene is average,
And the frame rate is low,
Then render all shadows via the depth-pass stencil shadow volume algorithm.

Rule #3
If there is an average number of light sources,
And the polygon complexity of the scene is high,
And the frame rate is medium,
Then render all shadows via the hardware-accelerated shadow mapping algorithm.

The first step of Mamdani inference is to "fuzzify" all precise input values via the definition of fuzzy sets. To do this, assume representation of the number of light sources through the range [0, 100] and definition of the following linguistic variables: `Few`, `Average` and `Many`. We can now define the following three fuzzy sets (the values are arbitrarily chosen):

$$F = \{(0,1),\ (50,0)\}$$

$$A = \{(30,0),\ (50,1),\ (70,0)\}$$

$$M = \{(50,0),\ (100,1)\}$$

A scene consisting of 60 light sources will, for example, result in the following fuzzy membership values for each of the three sets (Figure 5.3 shows the membership functions for these three fuzzy sets):

$$M_F(60) = 0$$

$$M_A(60) = 0.5$$

$$M_M(60) = 0.2$$

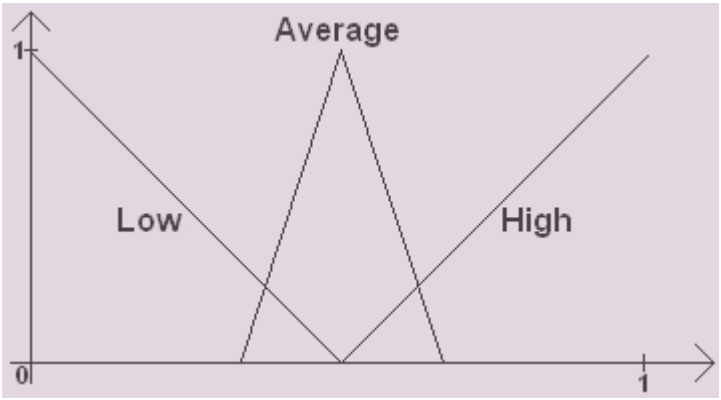


Fig 5.3 Membership functions.

We also have to factor in the frame rate performance and polygonal complexity of the scene being rendered. Both of these will have a number of membership functions and linguistic variables (`Low`, `Average` and `High`, respectively). For instance, our frame rate performance, ranging from 0 to 80, is given by the following membership functions (the values were again arbitrarily chosen):

$$L = \{(0,1),\ (30,0)\}$$
$$A = \{(20,0),\ (40,1),\ (60,0)\}$$
$$H = \{(50,0),\ (80,1)\}$$

A scene with a frame rate of 28 frames per second will, for example, result in the following fuzzy membership values for each of the three sets (via linear extrapolation):

$$M_L(28) = 0.067$$
$$M_A(28) = 0.7$$
$$M_H(28) = 0$$

All that remains now are the polygonal complexity membership functions ranging from 0 to 1000 (defined using the previous `Low`, `Average` and `High` linguistic variables):

$$L = \{(0,1),\ (600,0)\}$$
$$A = \{(200,0),\ (500,1),\ (800,0)\}$$
$$H = \{(600,0),\ (1000,1)\}$$

A scene with a polygonal complexity of 550 will, for example, result in the following fuzzy membership values for each of the three sets:

$$M_L(550) = 0.083$$
$$M_A(550) = 0.83$$
$$M_H(550) = 0$$

These fuzzy values can now be applied to the previously listed rules. For example, the first rule deals with the number of light sources. Hence, a scene featuring an average number of light sources (60) will result in a membership value of 0.5. Next we can factor in rule 2:

> If there are many light sources,
> And the polygon complexity of the scene is average,
> And the frame rate is low,
> Then render all shadows via the depth-pass stencil shadow volume algorithm.

Rule 2's membership values are given here:

$$M_M(60) = 0.2 \ \ (light \ \ sources)$$
$$M_A(550) = 0.83 \ \ (average \ \ polygon \ \ count)$$
$$M_L(28) = 0.067 \ \ (frame \ \ rate)$$

When working with multiple fuzzy variables, we always take the minimum membership value of a conjunction as the fuzzy value. Rule 2, governing the rendering of shadows using the depth-pass stencil shadow volume algorithm, will thus have a fuzzy membership value of 0.067.

Rule 3 can be evaluated similarly, resulting in the following membership values:

$$M_A(60) = 0.5 \ \ (light \ \ sources)$$
$$M_H(550) = 0 \ \ (polygon \ \ count)$$
$$M_M(28) = 0 \ \ (frame \ \ rate)$$

Rule 3, governing the rendering of shadows via the hardware-accelerated shadow mapping algorithm, will thus have a fuzzy membership value of 0.

For example, say we take our three fuzzy values as: *0.5* for rule 1 (to render all shadows via the hardware-accelerated shadow mapping algorithm), *0.067* for rule 2 (to render all shadows via the depth-pass stencil shadow volume algorithm) and *0* for rule 3 (to render all shadows via the hardware-accelerated shadow mapping algorithm).

The next step is to combine the calculated fuzzy values. We thus have 0.5 (rule 1) and 0 (rule 2) for hardware shadow mapping and 0.067 for stencil shadow volumes. The easiest method to combine these values is summation – giving us 0.5 for hardware shadow mapping and 0.067 for stencil shadow volumes.

The final step is *defuzzification* – the calculation of an exact value from a set of fuzzy values. Defuzzification can be performed by calculating the centroid of the area defined by a set of fuzzy values. A *centroid* is the intersection of all the lines dividing an object into equal parts. The centroid of an object can be considered its centre of mass. Figure 5.4 shows the centroid of a triangle.

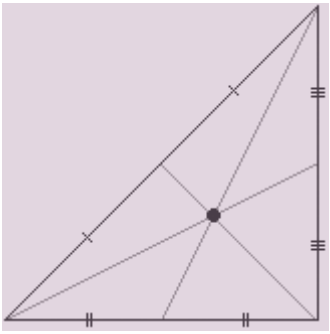


Figure 5.4 The centroid of a triangle.

We can calculate the centroid (the exact system output value) of an area using the following formula:

$$Centroid = \frac{\sum M(x)x}{\sum M(x)}; with\ M(x),\ the\ inclusive\ member\ function\ area.$$

Thus, the centre of gravity of the combined fuzzy output (the area created by clipping the membership function for hardware shadow mapping at 0.5 and for stencil shadow volumes at 0.067) can be calculated as follows:

$$Centroid = \frac{(5\times 0.067)+(10\times 0.1)+(15\times 0.15)+(20\times 0.2)+...+(100\times 1)}{0.067+0.1+0.15+0.2+...+1}$$

$$= \frac{717.585}{10.517}$$

$$= 68.2$$

The calculated crisp value of 68.2 indicates the amount, on a scale of 0 to 100, by which selection of the hardware shadow mapping algorithm is certain.

Mamdani inference is thus an integral part of fuzzy expert systems, with the functionality of these expert systems summarised as follows:

1) Load expert data.
2) Define the fuzzy sets and rules.
3) Associate observed data with the fuzzy sets.
4) Run through each case for each and every fuzzy rule.
5) Calculate the rule-based fuzzy values.
6) Combine the calculated fuzzy values.
7) Calculate an exact value from the set of fuzzy values.

Our fuzzy logic based expert system uses Mamdani inference to apply a set of fuzzy rules on a set of traditional precise inputs to obtain a precise output value, specifically an action recommendation (the shadow algorithm to utilise).

5.4 Summary

In this chapter we focussed on expert systems as an AI technique often employed to store and access human expertise. We also looked at fuzzy logic based expert systems and the combination of production rules with fuzzy logic to explicitly symbolise data.

We started by discussing expert systems as an artificial knowledge system defined by means of training, empirical experimentation and prior human experience. We also described a functioning expert system as a collection of elements, specifically as a system combining a set of facts, the relationship among these facts and a mechanism for the storage and rapid retrieval of this information.

Following this introduction, we investigated basic expert system architecture and two decision making techniques, namely forward chaining and backward chaining. We also looked at the advantages inherent to the use of expert systems.

The final section dealt with fuzzy reasoning and fuzzy expert systems. It specifically focussed on fuzzy logic as an extension to traditional predicate/Boolean logic and fuzzy sets as an extension to traditional sets. The concept of membership functions and their use in the definition of fuzzy sets as well as Mamdani inference was also presented in detail.

In the next chapter, we apply these techniques in the discussion of our empirically derived system for high-speed shadow rendering.

Chapter 6

An Empirically Derived System for High-Speed Shadow Rendering

Chapter 6 discusses the previously mentioned expert system implementation in much more detail. It also presents the critical analysis of our empirically derived system for the high-speed rendering of shadows. This analysis highlights not only the performance benefits inherent to the utilisation of this system, but also the practicality of such an implementation.

In this chapter we will investigate:

- Expert systems and dynamic shadow selection
- Rules for selection of shadow rendering algorithms
- Fuzzy rules for selection of the most appropriate shadow rendering algorithm
- Mamdani implementation
- Construction of the algorithm selection mechanism
- Results obtained from our benchmarking environment

6.1 Introduction

An expert system, as mentioned, analyses a dataset to determine the best solution to a given problem. Our real-time shadow generation framework employs such a system. This system specifically consists of an empirically ascertained dataset, a collection of rules to analyse the data and information of various elements pertaining to the scene currently being rendered.

We use an expert system, as previously discussed, to control the real-time selection of shadow rendering algorithms. The knowledge base of this expert system consists of experimental results obtained through the critical analysis of numerous real-time shadow rendering algorithms (and the improvements made through the use of various hardware extensions and hybrid approaches). Production rules are implemented via an inference engine. This inference engine is in turn used to select the most appropriate algorithm based on certain properties of the scene being rendered. For instance, our system could contain the following production rule: if there are a lot of light sources in a scene and the scene has a high geometric complexity, then enable a hybrid stencil shadow volume/shadow mapping algorithm. The notions “a lot of light sources” and “high geometric complexity” are not quantitative facts. Fuzzy logic provides a solution to this problem by assigning quantitative values and/or ranges to these concepts. The concepts “a lot of light sources” and “high geometric complexity” can also be combined into the new one “overly complex”, resulting in a new production rule. Our framework combines production rules with fuzzy logic to explicitly symbolise data. This is followed by the selection of the most efficient shadow rendering algorithm.

We will now look at the previously mentioned expert system implementation in much more detail. We will also perform a critical analysis of our empirically derived system for the high-speed rendering of shadows. This analysis will convey not only the performance benefits inherent to the utilisation of this system, but also the practicality of such an implementation.

6.2 Expert Systems and Dynamic Shadow Selection

Our empirically derived system for high-speed shadow rendering consists of a fuzzy logic based expert system and several shadow rendering algorithms. The expert system controls, as mentioned, the selection of these algorithms by correlating the properties of the scene being rendered with the previously obtained algorithmic performance data.

The expert system consists of the following modules:

- An inference engine.
- A fact database.
- A knowledge base.
- An explanation/debugging system.

The expert system's knowledge base consists of experimental results obtained through the critical analysis of numerous real-time shadow rendering algorithms (and the improvements made through the use of various hardware extensions and hybrid approaches). Production rules are implemented via an inference engine. This inference engine is in turn used to select the most appropriate algorithm based on certain properties of the scene being rendered. The knowledge base, as previously mentioned, is nothing more than a database of rules. These rules symbolise the stored knowledge. The fact database embodies the expert system inputs which are subsequently used to make decisions and/or to take certain actions (properties and statistics of the scene being rendered). The inference engine makes the actual decision by combining these expert system rules and facts. The explanation system generates information about the manner in which a decision was made. Figure 6.1 illustrates the architecture of our expert system.

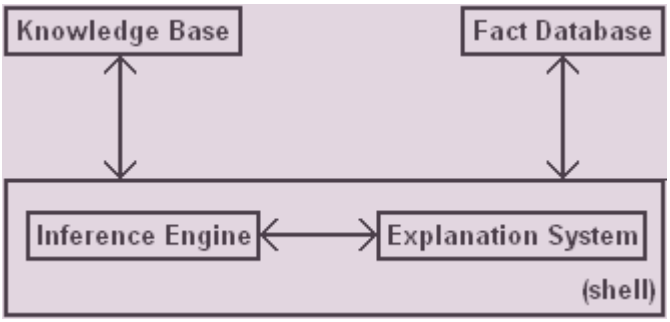


Fig 6.1 Architecture of our expert system.

Our expert system's inference engine and explanation system are contained within a "shell". The knowledge base and fact database are connected to this shell in a plugin-like fashion. The expert system shell is used to define a generic expert system, with the expert system's functionality controlled by the connected fact database and knowledge base.

Our expert system implementation utilises a forward chaining strategy to determine results from a collection of rules and facts. We basically start by reading the expert system inputs from the fact database followed by a comparison between the read inputs and the rules within the rule database. Now, if an input fact matches all the antecedents of a rule, then the rule is triggered with its conclusion added to the fact database.

Shadow selection is based on the maximisation of the rendering frame rate and shadow quality. The implemented expert system will thus select shadow generation algorithms by taking not only the scene's frames per second performance data into account but also by factoring in the viewer's position in relation to the shadow being rendered. The rendering accuracy and detail of distant shadows will thus carry less weight than those rendered relatively close to the viewer. Table 6.1 summarises the algorithms of choice based on the algorithmic comparison given in Section 3.5 and scene conditions such as view distance, dynamic/static light conditions and number of light sources.

Most Appropriate Algorithm	Conditions
The spatial subdivision approach coupled with SSE2/3DNow! utilisation.	All environmental areas lit using static light sources.
Chan and Durand's (2004) algorithm.	Scenes consisting of eight or less dynamic light sources when high-quality shadows are required and where shadow casting objects are located near the point-of-view.
Shadow mapping.	Scenes consisting of more than two and less than fourteen dynamic light sources and where the shadow casting objects are located a significant distance from the point-of-view.
	Chan and Durant's algorithm will, however, prove a better choice for both close range and distant objects when rendering scenes consisting of fourteen or more dynamic light sources. We will also use Chan and Durrand's algorithm for all scenes consisting of nine or more dynamic light sources when high-quality shadows are required.
McCool's (2000) and Thakur et al's (2003) algorithm.	The second best choice when dealing with scenes featuring one to eight light sources and when high-quality shadows are required. We won't be utilising this algorithm, rather opting for Chan and Durand's hybrid approach. The same goes for the classic stencil shadow volume algorithm and Thakur et al's (2003) algorithm.

Table 6.1 Algorithms of choice based on the analysis given in Section 3.5.

Returning to our expert system, we can create the following rules for selection of the most appropriate shadow rendering algorithm (these rules are derived from the algorithmic comparison given Section 3.5 and Table 4.1):

Rule #1
If the environment/sub-environment consists of only static light sources,
Then render all shadows via our spatial subdivision/SSE2/3DNow! algorithm.

Rule #2
If the scene consists of eight or less dynamic light sources,
And high-quality shadows are required (shadow casting objects are located near the point-of-view),
Then render all shadows via Chan and Durand's (2004) algorithm.

Rule #3
If the scene consists of more than two and less than fourteen dynamic light sources,
And low-quality shadows are required (shadow casting objects are located a significant distance from the point-of-view),
Then render all shadows via the basic Shadow mapping algorithm.

Rule #4
If the scene consists of fourteen or more dynamic light sources,
And either low- or high-quality shadows are required (for both close range and distant objects),
Then render all shadows via Chan and Durand's (2004) algorithm.

Rule #5
If the scene consists of nine or more dynamic light sources,
And high-quality shadows are required (shadow casting objects are located near the point-of-view),
Then render all shadows via Chan and Durand's (2004) algorithm.

A subsequent aspect of our expert system implementation is its fuzzy logic-based nature. As a fuzzy logic based expert system, it utilises a set of linguistic variables (related to the problem) and several membership functions. Fuzzy rules are derived from these variables as well as the knowledge base. These rules are applied by means of Mamdani fuzzy inference. Mamdani inference, as previously mentioned, applies a set of fuzzy rules on a set of traditional precise inputs to obtain a precise output value (such as an action recommendation).

Our fuzzy logic based expert system will thus contain the following fuzzy rules for selection of the most appropriate shadow rendering algorithm (these rules are screen resolution independent, lower resolutions will simply imply faster overall graphics performance with the shadow generation phases remaining consistent):

Rule #1
If the environment/sub-environment consists of stationary light sources,
Then render all shadows via our spatial subdivision/SSE2/3DNow! algorithm.

Rule #2
If the scene consists of an average number of dynamic light sources,
And high-quality shadows are required (shadow casting objects are located near the point-of-view),
Then render all shadows via Chan and Durand's (2004) algorithm.

Rule #3
If the scene consists of few or and less than an above average number of dynamic light sources,
And low-quality shadows are required (shadow casting objects are located a significant distance from the point-of-view),
Then render all shadows via the basic Shadow mapping algorithm.

Rule #4
If the scene consists of many dynamic light sources,
And either low- or high-quality shadows are required (for both close range and distant objects),
Then render all shadows via Chan and Durand's (2004) algorithm.

Rule #5
If the scene consists of an average or greater than average number of dynamic light sources,
And high-quality shadows are required (shadow casting objects are located near the point-of-view),
Then render all shadows via Chan and Durand's (2004) algorithm.

We use Mamdani inference, as explained in Chapter 5, to "fuzzify" all precise input values via the definition of fuzzy sets. As in our Chapter 5 example, we assume a representation of the number of light sources through the range [0, 20], the nature of a scene's light sources via the values 1 for dynamic and 0 for static and the distance from the viewer via the range [0, 90] (in world units). Our implementation also defines the following linguistic variables: `Stationary`, `Dynamic`, `Average`, `Few` and `Many`. The system is thus based on Mamdani inference to apply a set of fuzzy rules on a set of traditional precise inputs to obtain a precise output value, specifically an action recommendation (the shadow algorithm to utilise).

Our Mamdani implementation will thus load the critical analysis performance data, read the pre-programmed fuzzy sets and rules, associate the observed data with the fuzzy sets, run through each case for each and every fuzzy rule, calculate the rule-based fuzzy values, combine the calculated fuzzy values and finally calculate an exact value from the set of fuzzy values.

6.3 Construction of the algorithm selection mechanism

Our algorithm selection mechanism basically consists of a fuzzy-logic based expert system and a number of shadow rendering algorithms. The knowledge base of this expert system, as previously discussed, consists of experimental results obtained through the critical analysis of numerous real-time shadow rendering algorithms (and the improvements made through the use of various hardware extensions and hybrid approaches). Production rules are implemented via an inference engine. This inference engine is in turn used to select the most appropriate algorithm based on

certain properties of the scene being rendered. Our framework combines production rules with fuzzy logic to explicitly symbolise data. This is followed by the selection of the most efficient shadow rendering algorithm.

The data gathered during the previously discussed critical analysis allows for the construction of a fuzzy logic-based expert system. This system, as mentioned, controls the real-time selection of shadow rendering algorithms based on environmental conditions. We basically store the gathered data in a comma-delimited format with the rendering framework loading it into memory upon execution (by utilising a number of sorted associative arrays or maps). Each implemented shadow rendering algorithm is, in turn, loaded into our framework via a dynamic link library. DLLs are based on Microsoft's shared library concept and can contain source code, data and resources. These libraries are generally loaded at runtime, a process referred to as run-time dynamic linking – thus allowing us to replace or change DLLs without recompiling the main executable. The shadow rendering DLL contains the implementation details of the basic stencil shadow volume algorithm, the basic hardware shadow mapping algorithm, McCool's shadow volume reconstruction using depth maps, Eric Chan and Frédo Durand's hybrid algorithm for the efficient rendering of hard-edged shadows, Thakur et al's algorithm based on the elimination of various shadow volume testing phases and our algorithm based on shadow volumes, spatial subdivision and instruction set utilisation.

6.4 Results

By dynamically cycling through shadow generation algorithms to compensate for performance-impacting changes in our rendering environment, we are able to bridge an existing gap between shadow quality and high-speed shadow rendering. We will now highlight the performance gains inherent to our system's use when compared to traditional approaches.

Our benchmarking environment consisted of an initial number of static light sources. We then added a number of dynamic light sources (six) with shadow casting objects positioned relatively close to the viewer. This allowed us to analyse the transition from the spatial subdivision/SSE2/3DNow! algorithm to Chan and Durand's (2004) algorithm. Following this we increased the number of dynamic light sources to thirteen with the shadow casting objects translated to a significant distance from the point-of-view. All shadows previously rendered using Chan and Durand's (2004) algorithm were now rendered via shadow mapping. Next we systematically increased the number of light sources to sixteen while leaving the shadow casting objects at their previous position – this caused a reselection of Chan and Durand's (2004) algorithm. The shadow casting objects were subsequently translated back to their previous position (relatively close to the viewer) with the scene's lighting reset to nine dynamic light sources (with shadow casting objects located near the point-of-view) – Chan and Durand's (2004) algorithm was successfully selected. Figure 6.2 shows the performance data obtained (for this specific instance) for up to eight light sources with Figure 6.3 showing the results obtained for nine to sixteen light sources.

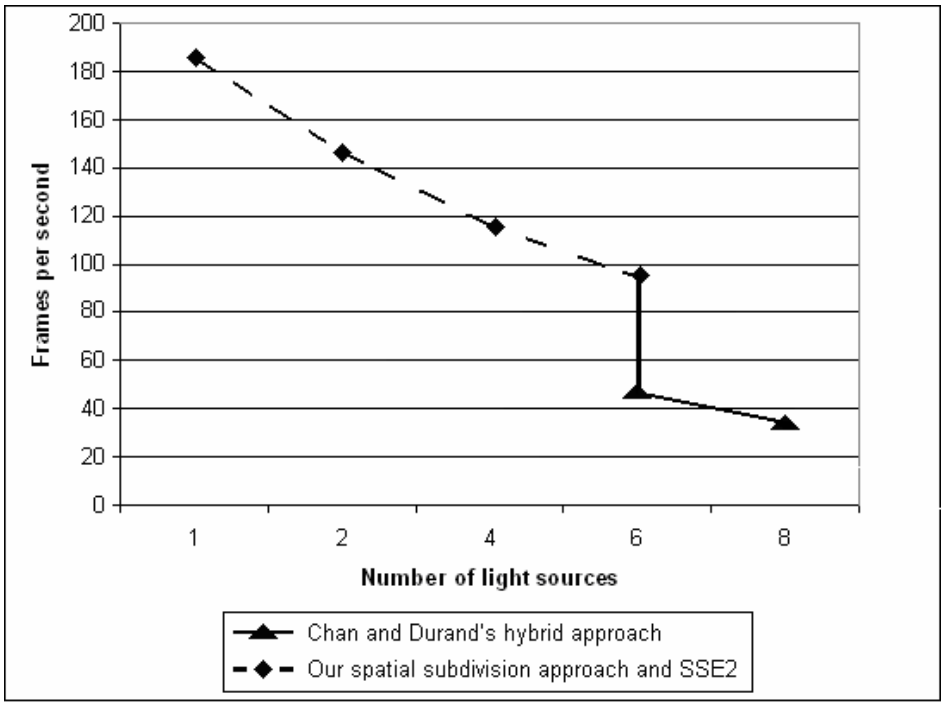


Figure 6.2 Performance data for up to eight light sources.

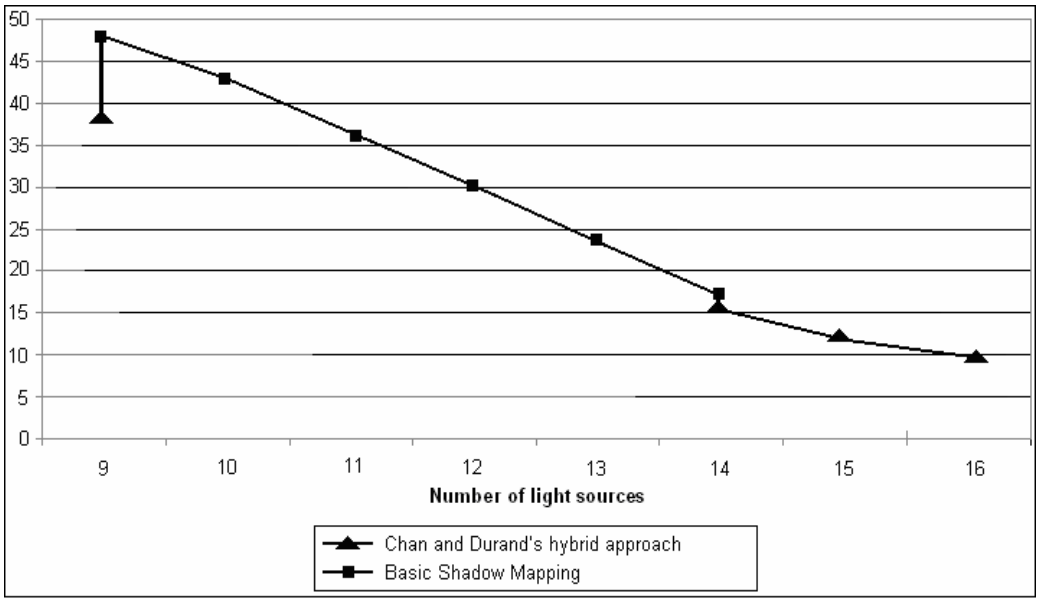


Figure 6.3 Performance data for nine to sixteen light sources.

We can repeat the experiment in reverse order – that is, by starting with nine dynamic light sources (with shadow casting objects located near the point-of-view). Our benchmarking environment will thus select Chan and Durand's (2004) algorithm as its initial shadow rendering algorithm. Next we systematically increased the number of light sources to sixteen while leaving the shadow casting objects at their previous position – Chan and Durand's (2004) algorithm will still be the algorithm of choice and no alternative shadow rendering algorithms will be selected. Following this we decreased the number of dynamic light sources to thirteen with the shadow casting objects translated to a significant distance from the point-of-view. All shadows previously rendered using Chan and Durand's (2004) algorithm were now rendered

via shadow mapping. We now decreased the number of dynamic light sources to six with the shadow casting objects positioned relatively close to the viewer. This allowed us to analyse the transition from Chan and Durand's (2004) algorithm to the spatial subdivision/SSE2/3DNow! algorithm. Our final action was to set all the dynamic light sources to static. Figure 6.4 shows the performance data obtained for sixteen to nine light sources with Figure 6.5 showing the results obtained for eight to a single light source. Figure 6.6 shows the interactive testing environment.

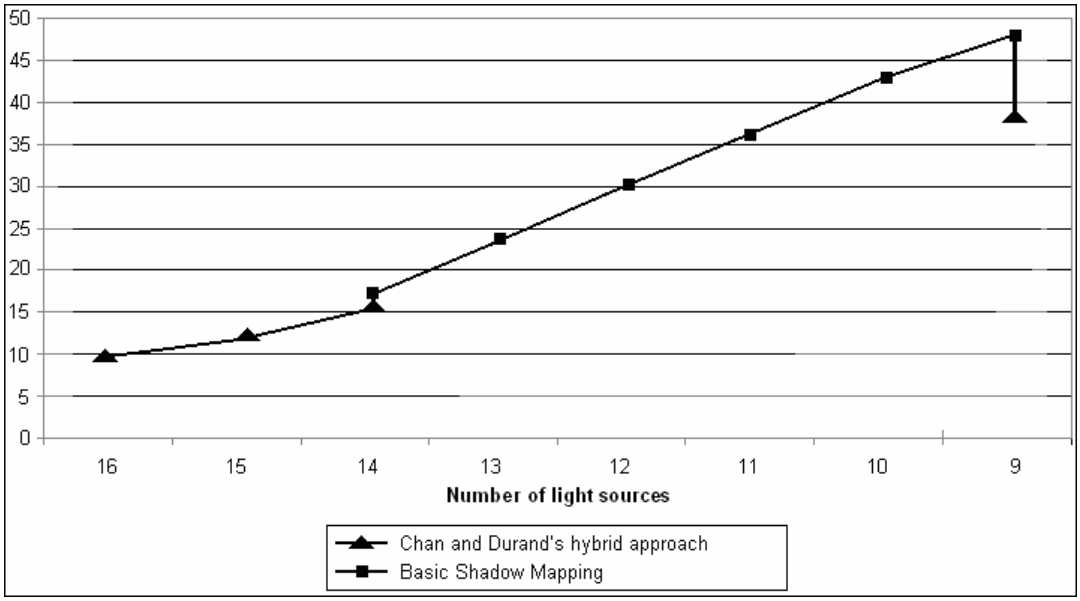


Figure 6.4 Performance data for sixteen to nine light sources.

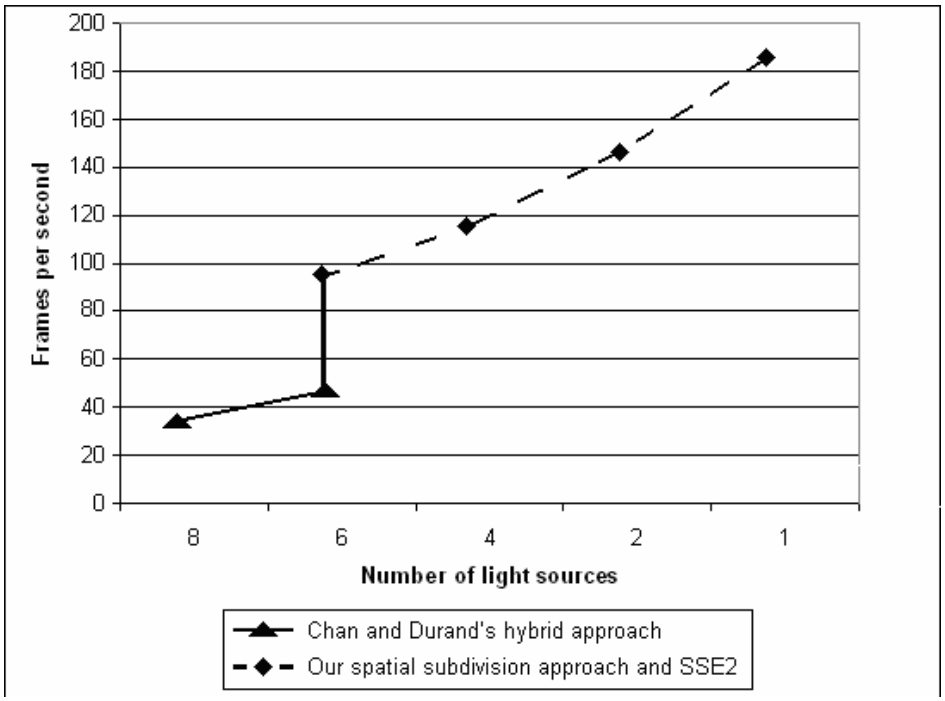


Figure 6.2 Performance data for eight to a single light source.

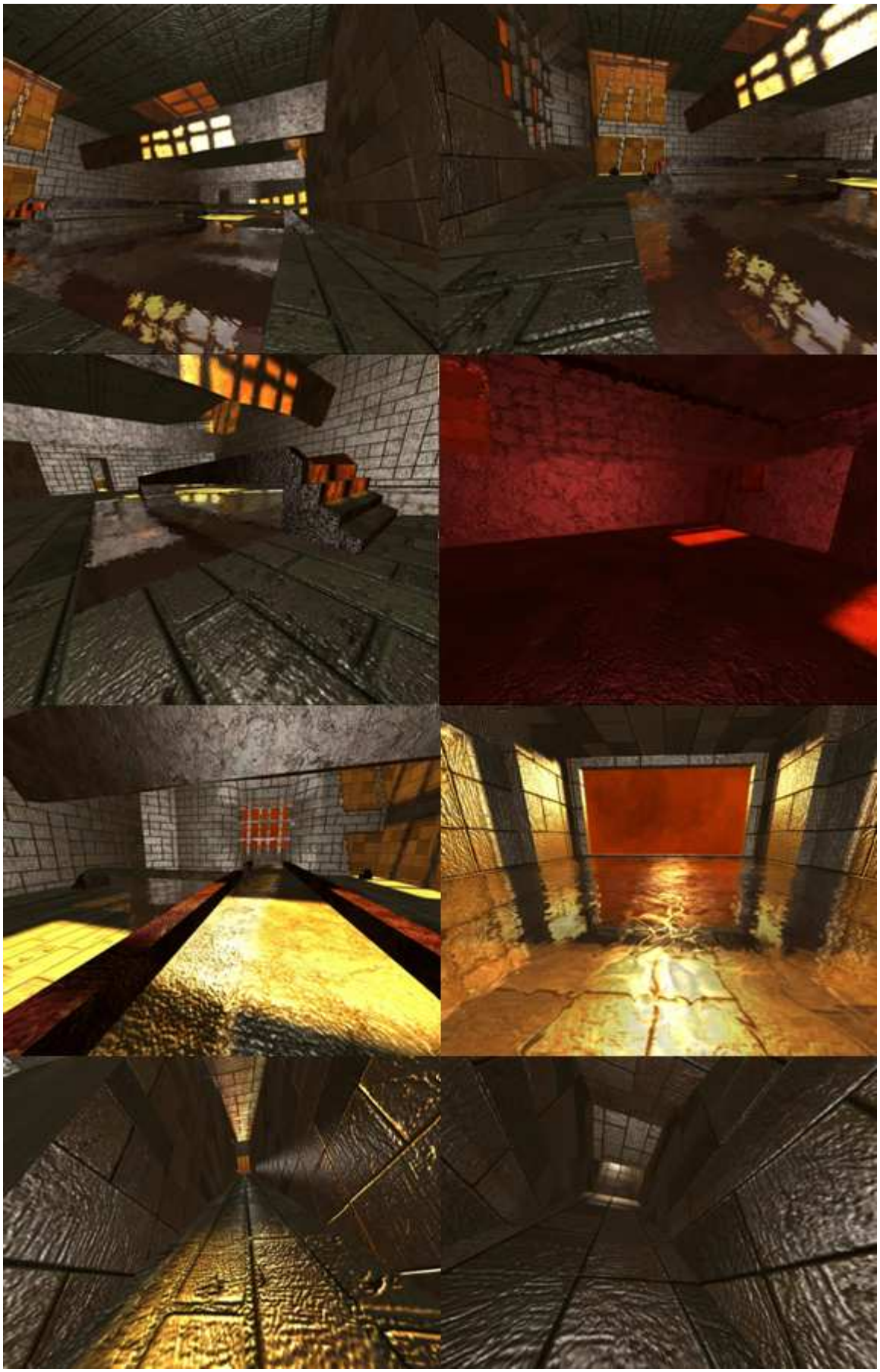


Figure 6.6 Various screenshots of our interactive testing environment.

6.5 Summary

This chapter presented the general architecture of our empirically derived system for high-speed shadow rendering – the expert system component being the main focus. We also looked at fuzzy logic-based reasoning for the explicit symbolisation of data.

The final section summarised the results obtained by dynamically cycling through shadow generation algorithms to compensate for performance-impacting changes in a rendering environment. These results illustrated the performance gains inherent to our system's use. The next chapter gives an overall summary of our work. It closes by discussing possible future work based on the presented research.

Chapter 7

Summary and Conclusion

Chapter 7 features an overall summary of our work. It closes by discussing possible future work based on the presented research.

In this chapter we will present:

- An overall summary of our work
- Concluding remarks

7.1 Summary

We initially focussed on the fundamentals of lighting and real-time shadow generation in detail. Light sources were introduced with regard to the role they play in the creation of properly lit and shaded objects. Building on this we presented the illumination function as a way to describe any light source in terms of six variables. We also discussed how a lighting model can be used to define light-object interactions based on the type of light source and the material properties of the object.

We subsequently investigated a number of light source types, specifically looking at point lights as light sources emitting light uniformly in 360 degrees, spotlights as point lights emitting light within an angle range, ambient lighting as a way to provide a uniform level of illumination throughout a scene, parallel lights as sources illuminating objects through a number of parallel light rays and emissive light as self-reflecting light originating from an object's surface.

Following this we looked at several reflection models, namely, the ambient reflection model, the specular reflection model, diffuse reflection and the Phong reflection model. We specifically investigated these reflection models as functions of material properties (e.g. surface reflectance, colour, etc) and light source properties (e.g. light direction, colour, position, attenuation, etc).

Next we investigated real-time shadow generation and its contribution to the realism and ambience of rendered environments. We started by defining a shadow as the two-dimensional projection of at least one object onto another object or surface, subsequently looking at the physical properties of shadows and the technique of shadow casting in general.

Following this introduction we investigated a number of shadowing algorithms, specifically scan-line polygon projection, Blinn's shadow polygons, shadow mapping and stencil shadow volumes. Our discussion of Blinn's shadow polygons revealed it as an extremely easy to use shadow generation technique often utilised to render the shadows of single polygons on flat surfaces. We also considered the quite complex, and now mostly redundant scan-line polygon projection algorithm historically used for the generation of hard-edged shadows.

We subsequently discussed the fundamentals of shadow mapping and stencil shadow volumes, our shadow mapping discussion highlighting the general dual-pass shadow mapping technique. The shadow volume section presented the theory behind the construction of finite shadow volumes as well as the stencilling process, differentiating between the depth-pass and depth-fail methods used to test whether a fragment is in shadow or not. We also briefly touched on the generation of soft-edged shadows using penumbra wedges.

Next we discussed the implementation details of various shadow rendering algorithms, specifically the stencil shadow volume algorithm, the shadow mapping

algorithm and a number of hybrid approaches such as McCool's shadow volume reconstruction using depth maps, Chan and Durand's hybrid algorithm for the efficient rendering of hard-edged shadows, Thakur et al's elimination of various shadow volume testing phases and Rautenbach et al's shadow volumes, hardware extensions and spatial subdivision approach. We also proceeded by presenting a stencil shadow volume benchmarking program consisting of a relatively simple static cubic environment, a movable/interchangeable three-dimensional mesh object and a variable number of light sources.

We then introduced the stencil buffer as a buffer used for controlling the rendering of selected pixels. The associated per-pixel test, namely stencilling, was also looked at in detail. We then investigated the depth-stencil testing process followed by a discussion detailing the shadow volume implementation (with specific focus being on the shadow volume construction process and calculation of silhouette edges).

Following this we looked at a shadow mapping evaluation environment as well as the implementation of shadow maps. Following this we looked at a shadow mapping evaluation environment as well as the implementation of shadow maps. We then dealt with the implementation details of various hybrid approaches such as McCool's shadow volume reconstruction using depth maps, Chan and Durand's hybrid algorithm for the efficient rendering of hard-edged shadows, Thakur et al's elimination of various shadow volume testing phases and Rautenbach et al's shadow volumes, hardware extensions and spatial subdivision approach.

Building on this, we presented our benchmarking mechanism and a set of criteria for the evaluation of shadow generation techniques. The given evaluation criteria were selected with the aim of assessing the relationship between shadow rendering quality and performance – in turn allowing us to isolate key algorithmic weaknesses and possible bottleneck areas. We also investigated several possibilities for improving the performance and quality of shadow rendering algorithms; both on a hardware and software level.

Specific algorithms benchmarked and analysed include: the basic stencil shadow volume algorithm, the basic hardware shadow mapping algorithm, McCool's shadow volume reconstruction using depth maps, Chan and Durand's hybrid algorithm for the efficient rendering of hard-edged shadows, Thakur et al's algorithm based on the elimination of various shadow volume testing phases and our algorithm based on shadow volumes, spatial subdivision and instruction set utilisation.

Our critical analysis concluded with a comparison between the results obtained; we also summarised these results with specific emphasis on the most appropriate application area.

Following this we focussed on expert systems as an AI technique often employed to store and access human expertise. We also looked at fuzzy logic based expert systems and the combination of production rules with fuzzy logic to explicitly symbolise data.

We started by discussing expert systems as an artificial knowledge system defined by means of training, empirical experimentation and prior human experience. We also described a functioning expert system as a collection of elements, specifically as a system combining a set of facts, the relationship among these facts and a mechanism for the storage and rapid retrieval of this information.

Following this introduction, we investigated basic expert system architecture and two decision making techniques, namely forward chaining and backward chaining. We also looked at the advantages inherent to the use of expert systems.

The final section of our expert system literature study dealt with fuzzy reasoning and fuzzy expert systems. It specifically focussed on fuzzy logic as an extension to traditional predicate/Boolean logic and fuzzy sets as an extension to traditional sets. The concept of membership functions and their use in the definition of fuzzy sets as well as Mamdani inference was also presented in detail.

The final chapter presented the general architecture of our empirically derived system for high-speed shadow rendering – the expert system component being the main focus. We also looked at fuzzy logic-based reasoning for the explicit symbolisation of data.

The dissertation closed by presenting the results obtained from dynamically cycling through shadow generation algorithms to compensate for performance-impacting changes in a rendering environment. These results illustrated the performance gains inherent to our system's use.

7.2 Concluding Remarks

The computer graphics industry has developed immensely during the past decade. Looking at the area of computer games one can easily see technological leaps being made on a yearly basis. However, most of the currently available shadow rendering algorithms are only amenable to specific rendering conditions and/or situations. For example, the "vanilla" depth-fail/depth-pass stencil shadow volume algorithm is based on a series of processor intensive conditionally executed branches with its silhouette detection and shadow volume construction process taking up a significant amount of processing time (especially in scenes where more light sources and shadow casting objects are added). We replaced these conditionally executed branches with SSE and 3DNow! instructions, forcing their parallel execution during each rendering cycle and achieving a significant performance increase. Considering this improvement, we went even further by combining the depth-fail stencil shadow volume algorithm with spatial subdivision. The implementation of this method resulted in significant performance gains, especially where a statically lit scene consisted of several concealed objects. That said; our own algorithm was also limited to a specific condition – static light sources.

The only viable solution was to perform a critical analysis of numerous shadow rendering algorithms with the aim of assessing the relationship between shadow rendering quality and performance – in turn allowing us to isolate key algorithmic weaknesses and possible bottleneck areas. Using this performance data, we were able to construct a fuzzy logic-based expert system to control the real-time selection of shadow rendering algorithms based on environmental conditions (as discussed in Chapter 6). This system ensured the following: nearby shadows were always of high-quality, distant shadows would, under certain conditions, be rendered at a lower quality and the frame rate would always run at a maximum. It is important to note that this framework is also adaptable for use with other rendering algorithms such as real-time reflections and particle systems. Adapting this framework for use on 3D capable mobile devices (such as the iPhone and iPod Touch) will give these devices the ability to render shadows (and other special effects) not previously possible without overburdening the processor.

We have also demonstrated the use of expert system technology as a viable solution to the problem of selecting between competing algorithms in real-time. This resulted in the reduction/optimisation of processor usage by ensuring that the quality of rendered shadows is appropriately tuned, for example, GPU/processor usage is never wasted to accurately render distant shadows.

It is also important to note that, despite all the algorithms available for the implementation of shadows, a lot of work remains in the field. More algorithms could, for example, be benchmarked and added to our expert system's knowledge base. Alternate algorithmic performance improvements can also be pursued.

Traditionally rendered shadows used in conjunction with AI subsystems, game networking and logic, physics processing and other rendering effects (such as real-time reflections, refraction, etc) is immensely processor intensive and can only be successfully implemented on high-end hardware. Only by cycling shadow algorithms based on environmental conditions and through the exploitation of algorithmic strengths can high-quality real-time shadow generation become as common as texture mapping.

References

Akenine-Möller T. and Assarsson U. (2002) Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges. *Proceedings of the 13th Eurographics Workshop on Rendering*. *Aire-la-Ville: Eurographics Association.*

ADVANCED MICRO DEVICES, INC. (2000) AMD 3DNow! Technology Manual. Published online at: www.amd.com.

Appel A. (1968) Some Techniques for Machine Rendering of Solids. *AFIPS Conference Proceedings*, 32.

Apple, INC. IPhone Technical Specifications. Published online at: http://www.apple.com/iphone/specs.html

Bell B. (2003) S3: From Virge to Savage 2000. Published online at: http://www.firingsquad.com/hardware/s3_deltachrome/default.asp

Bergeron, P. (1985) Shadow volumes for non-planar polygons. *Canadian Information Processing Society Graphics Interface 1985*, 417-418, (SEE N85-34523 23-61), Canada.

Blinn J. (1988) Me and My (Fake) Shadow. *IEEE Computer Graphics and Applications*, 8(1):82-86.

Bouknight W. and Kelly K. (1970) An Algorithm for Producing Half-tone Computer Graphics Presentations with Shadows and Moveable Light Sources. *Proceedings of the AFIPS, Spring Joint Computer Conference*, 36.

Boulanger K., Pattanaik S. and Bouatouch K. (2006) Rendering Grass in real-time with Dynamic Light Sources and Shadows, ISSN: 1166-8687. *Technical Report no. 1809, July, IRISA, Rennes, France*.

Brabec S. and Seidel H. (2002) Single sample soft shadows using depth maps. *Graphics Interface*.

Brotman L.S. and Badler N.I. (1984) Generating Soft Shadows with a Depth Buffer Algorithm. *IEEE Computer Graphics and Applications*, 4(10):5-12.

Campbell-Kelly M. (2006) Edsac Simulator: An emulator of the EDSAC, including the code for OXO. Published online at: http://www.dcs.warwick.ac.uk/~edsac/

Carmack J. (2000) Carmack on shadow volumes. *Personal correspondence between Mark Kilgard and John Carmack*.

Chan E. and Durand F. (2004) An Efficient Hybrid Shadow Rendering Algorithm. *Proceedings of the Eurographics Symposium on Rendering*, 185-195.

Chen C. H. (1996) The Fuzzy Logic and Neural Network handbook. ISBN: 0-07-011189-8

Choppin B. (2004) Artificial Intelligence Illuminated, ISBN-13: 978-0763732301. *Jones & Bartlett Publishers; 1 edition*.

Craddock D. (2007) Alex St. John Interview. Published online at: http://www.shacknews.com/featuredarticle.x?id=283

Crow F. (1977) Shadow Algorithms for Computer Graphics. *SIGGRAPH Proceedings 1977*. New York: ACM.

Dimitrov R. (2007) Cascaded Shadow Maps. Published online at: www.developer.nvidia.com.

Drettakis G. and Fiume E. (1994) A fast shadow algorithm for area light sources using backprojection. *Computer Graphics (SIGGRAPH 1994)*, *Annual Conference Series, ACM SIGGRAPH*, pp. 223–230.

Everitt C., Rege A. and Cebenoyan C. (2001) Hardware Shadow Mapping. NVIDIA white paper published online at: http://developer.nvidia.com/object/hwshadowmap_paper.html.

Everitt C. and Kilgard M. (2002) Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. NVIDIA white paper published online at: http://developer.nvidia.com/object/ robust_shadow_volumes.html.

Fernando R., Fernandez S., Bala K. and Greenberg D. (2001) Adaptive shadow maps. *Computer Graphics (SIGGRAPH 2001), Annual Conference Series*, *ACM SIGGRAPH*, 387–390.

GameSpy (2001) GameSpy's Top 50 Games of All Time. Published online at: http://archive.gamespy.com/articles/july01/top501aspe/index4.shtm

Giarratano J., Riley G. (2005) Expert Systems, Principles and Programming, ISBN 0-534-38447-1

Haines E. (2001) Soft planar shadows using plateaus. *Journal of Graphics Tools*, 6(1):19–27.

Harbour J.S. (2004) Game Programming All in One (second edition), ISBN: 1598632892, *Boston, MA: Thomson Course Technology*.

Heidmann T. (1991) Real shadows real time. *IRIS Universe*, 18:28–31.

Heidrich W., Brabec S. and Seidel H. (2000) Soft shadow maps for linear lights high-quality. *Rendering Techniques 2000 (11th Eurographics Workshop on Rendering)*, Springer-Verlag, 269–280.

Hourcade J.-C. and Nicolas A. (1985) Algorithms for antialiased cast shadows. *Computers & Graphics*, 9(3):259–265.

Ignizio J. (1991) Introduction to Expert Systems, ISBN 0-07-909785-5

Intel. (2002) Getting Started with SSE/SSE2 for the Intel® Pentium® 4. Published online at: www.intel.com/cd/ids/developer/asmo-na/eng/popular/20240.htm

Kilgard M. J. (1999) Improving shadows and reflections via the stencil buffer. Published online at: www.developer.nvidia.com.

Kersten D., Mamassian P. and Knill D. (1994) Moving cast shadows and the perception of relative depth. *Technical Report no 6, Max-Planck-Institut fuer biologische Kybernetik*.

Kersten D., Mamassian P. and Knill D. (1997) Moving cast shadows and the perception of relative depth. *Perception*, 26(2):171–192.

Kirsch F. and Doellner J. (2003) Real-time soft shadows using a single light sample. *Journal of WSCG (Winter School on Computer Graphics 2003)*, 11(1).

Klietz A. (1992) Scepter - the first MUD? Published online at: http://groups.google.com/group/rec.games.mud/msg/e423bcf6cf93d73b?pli=1

Knight G. (2003) The Twists and Turns of the Amiga Saga. Published online at: http://www.amigahistory.co.uk/ahistory.html

Kolic I., Mihajlovic Z., Budin L. (2004) Stencil shadow volumes for complex and deformable objects. *Proceedings of the 2004 11th IEEE International Conference on 13-15 Dec. 2004*, 314–317

Kushner, D. (2003) Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture, ISBN 0-375-50524-5. *Random House*.

Lauritzen A. (2006) Variance Shadow Maps. Published online at: www.developer.nvidia.com.

Lokovic T. and Veach E. (2000 Deep shadow maps. *Computer Graphics (SIGGRAPH 2000), Annual Conference Series, ACM SIGGRAPH*, 385–392.

Mamdani E. H., Assilian S. (1975) An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller. *International Journal of Man-Machine Studies, 7, 1, 1-15, Jan 75*

McCool M. D. (2000) Shadow volume reconstruction from depth maps, ISSN 0730-0301. *ACM Transactions on Graphics 19, 1 (January), 1–26*.

Microsoft. (1995) Microsoft's Judgement Day video Promoting Windows 95 as a Platform that could deliver Cutting-edge Multimedia Experiences like Doom. Available online at: http://home.comcast.net/%7Ereelsplatter/BillDoomTitles.wmv (also provided on the back cover CD).

Miller M. (2005) A History of Home Video Game Consoles. Published online at: http://www.informit.com/articles/article.aspx?p=378141

Nguyen H. (2007) GPU Gems 3, ISBN: 0321515269. *Reading, MA: Addison-Wesley*.

Nilsson J. (1986) Principles of Artificial Intelligence, ISBN-13: 978-0934613101. *Morgan Kaufmann Publishers*.

Office of Scientific and Technical Information (OSTI). (1981) Video Games – Did They Begin at Brookhaven? Published online at: http://www.osti.gov/accomplishments/videogame.html

Ortutay B. (2008) Take-Two's 'Grand Theft Auto IV' tops $500M in week 1 sales". Associated Press. *Retrieved on 2008-05-08*.

Pharr M. and Fernando R. (2005) GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, ISBN: 0321335597. *Reading, MA: Addison-Wesley*.

Phong B. (1975) Illumination for Computer-Generated Pictures. *Communications of the ACM*, 18(6).

Powell, J. (1985) ST Product News: First ST review. Published online at: http://www.atarimagazines.com/v4n6/STproductnews.html

Rabin S. (ed.) (2005) Introduction to Game Development. ISBN: 1584503777. *Hingham, MA: Charles River Media*.

Rau S. (2002) AMD PR Rating. Published online at: www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/AMD_White_Paper_-_Final_Version_11.15.02.pdf

Rautenbach P. (2008) 3D Game Programming using DirectX 10 and OpenGL, ISBN-13: 978-1-84480-877-9. *Cengage Learning (formerly Thomson Learning EMEA), London*.

Rautenbach P., Pieterse V., Kourie D., (2008) Stencil Shadow Volume Algorithms: An Analysis and Enhancement. *9e Colloque Africain sur la Recherche en Informatique et en Mathématiques Appliquées (October)*.

Reeves W., Salesin D. and Cook R. (1987) Rendering antialiased shadows with depth maps. *Computer Graphics (SIGGRAPH 1987)*, 21(4):283–291.

Reimer J. (2005). Total share: 30 years of personal computer market share figures. Published online at: http://arstechnica.com/articles/culture/total-share.ars/4

Salton G. (1987) Expert systems and information retrieval. *SIGIR Forum* 21:3-4, 3-9.

Segal M., Korobkin C., van Widenfelt R., Foran J. and Haeberli P. (1992) Fast shadows and lighting effects using texture mapping. *Computer Graphics (SIGGRAPH 1992)*, 26(2):249–252.

Taylor A. (1982) Pac-Man Finally Meets His Match. Published online at: http://www.time.com/time/magazine/article/0,9171,923197,00.html

Thakur K., Cheng F. and Miura K.T. (2003) Shadow generation using discretized shadow volume in angular coordinates. *Computer Graphics and Applications Proceedings. 11th Pacific Conference on 8-10 Oct. 2003,* 224-233

Weyhrich S. (2001) Apple II History. Published online at: http://apple2history.org/history/ah03.html

Whitted T. (1980) An Improved Illumination Model for Shaded Display. *Communications of the ACM.* 23(6): 343-349.

Williams L. (1978) Casting Curved Shadows on Curved Surfaces. *Computer Graphics*, 12(3).

Winter D. (2004) PONG-Story: A.S. Douglas' 1952 Noughts and Crosses game. Published online at: http://www.pong-story.com/1952.htm

Yarusso A. (2007) AtariAge - 2600 Consoles and Clones. Published online at: http://www.atariage.com/2600/archives/consoles.html

Zadeh L. (1965) Fuzzy sets, Information Control 8, 338-353

Zadeh L. (1998) Knowledge representation in fuzzy logic. *IEEE Transactions on Knowledge and Data Engineering 1*, 89-100.