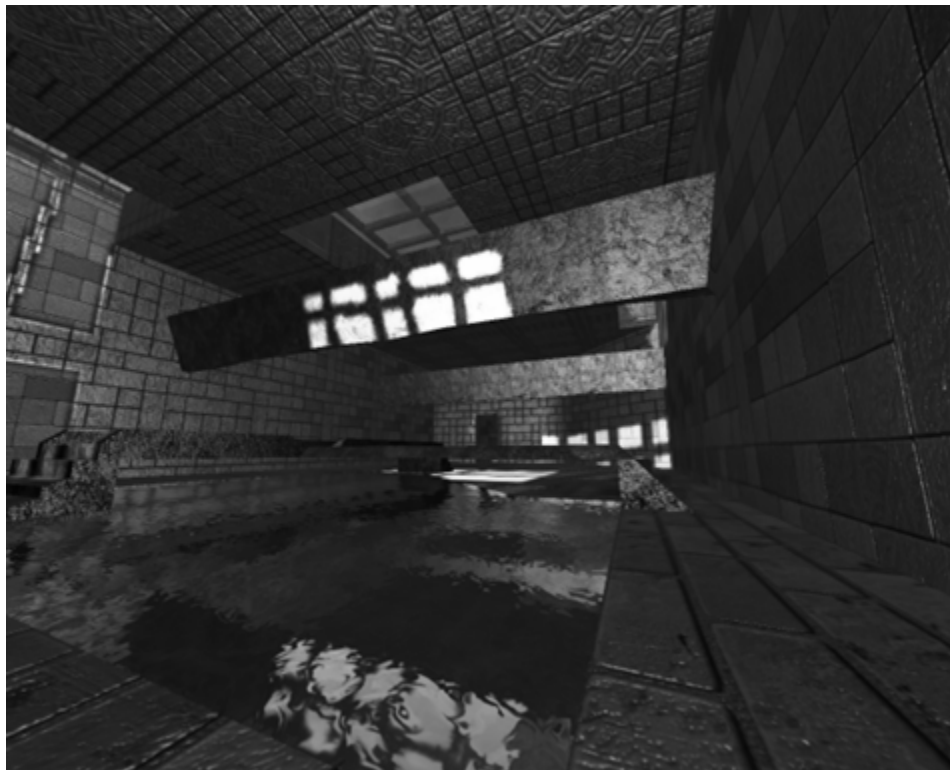


Table of Contents

Abstract	ii
Acknowledgements	v
Preface	vi
Part I	1
Chapter 1: Introduction	2
1.1 Research Domain	3
1.2 Problem Statement	12
1.3 Dissertation Structure	20
Chapter 2: Creating an Interactive 3D Environment	23
2.1 Game Engine Architecture	24
2.2 Initialisation and Shutdown	29
2.3 The Game Loop	30
2.4 Creating a Basic Interactive DirectX 10 3D Environment	33
2.5 Summary	47
Chapter 3: Extending the Basic Interactive 3D Environment	48
3.1 Extending the Basic Interactive DirectX 10 3D Environment	49
3.2 Shaders	51
3.3 Local Illumination	54
3.4 Reflection and Refraction	61
3.4.1 Implementing Cube Mapping	63
3.4.2 Implementing Basic Refraction	71
3.4.3 Reflection and Refraction Extended	76
3.5 Adding High Dynamic Range (HDR) Lighting	82
3.6 Shadows	87
3.6.1 Stencil Shadow Volumes	88
3.6.2 Implementing Shadow Mapping	103
3.6.3 Hybrid and Derived Approaches	105
3.7 Physics	108
3.7.1 The Role of Newton's Laws	109
3.7.2 Particle Effects	112
3.7.3 Particle System Implementation	113
3.8 Post-Processing	117
3.9 Summary	117
Part II	118
Chapter 4: Benchmarking the Rendering Algorithms and Techniques	119
4.1 Benchmarking Mechanism	120
4.2 Rendering Subsystem Evaluation Criteria	120
4.3 Algorithm Comparison	121
4.3.1 Shadows	122
4.3.2 Shaders	127
4.3.3 Local Illumination	131
4.3.4 Reflection and Refraction	134
4.3.5 Physics	137

4.3.6 Particle Effects	140
4.3.7 Post-Processing	144
4.4 Summary	147
Chapter 5: An Empirically Derived System for Distributed Rendering	149
5.1 Introduction	150
5.2 The Selection Engine and the Dynamic Selection and Allocation of Algorithms	150
5.2.1 Shadows	152
5.2.2 Shaders	154
5.2.3 Local Illumination	155
5.2.4 Reflection and Refraction	155
5.2.5 Physics	156
5.2.6 Particle Effects	157
5.2.7 Post-Processing	158
5.3 Construction of the Algorithm Selection Mechanism	158
5.4 Results	160
5.5 Summary	171
Chapter 6: Summary and Conclusion	173
6.1 Summary	174
6.2 Concluding Remarks and Future Work	176
References	179
Appendix A: Fundamentals of the Graphics Pipeline Architecture	200
Appendix B: Shaders	219
Appendix C: Lighting and Reflection	246
Appendix D: Real-time Shadow Generation	260
Appendix E: Physics	276
Appendix F: The DXUT Framework	294





Part I

Introduction and Implementation

Introduction

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

Outline:

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

1.1 Research Domain

In order to contextualise high-performance 3D rendering and engine design within its historical context, this chapter starts by offering a brief overview of computer gaming – the primary driving force behind the continued advancement of real-time rendering systems such as the one developed for this thesis. Note that 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.

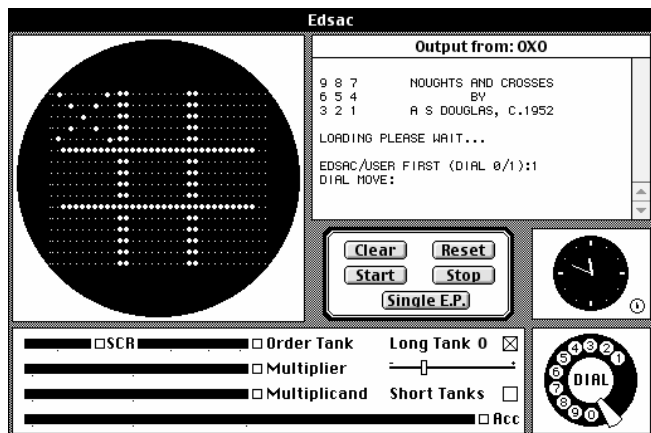


Figure 1.1 A screenshot of the game *OXO*.

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.

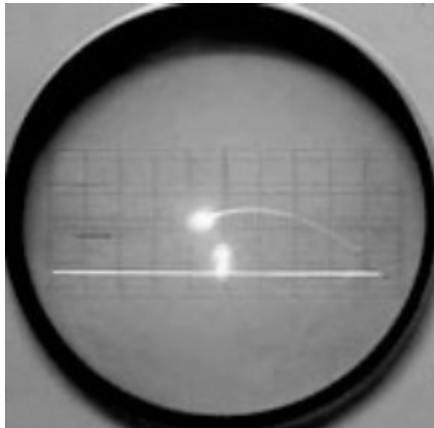


Figure 1.2 A photograph of the game Tennis for Two.

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 (Massively Multiplayer Online Games) 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, but was based on the sport of table tennis and had 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.

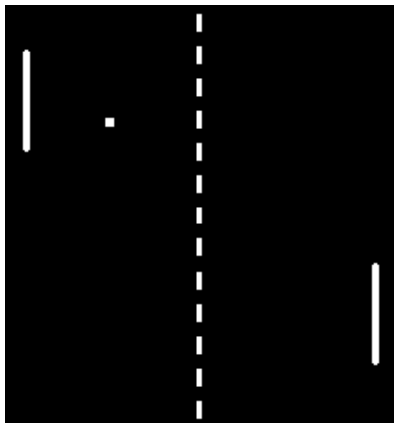


Figure 1.3 A PONG clone.

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 to learn 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 (an example is shown in Figure 1.6) were priced out of the reach of most home users 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 was the primary reason for the growth in popularity of the PC at that stage. 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 significantly drove the adoption of computer gaming but it was not 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 *Hovortank 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 1.8 *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 was 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 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 are defined as a series of specifications that are, in turn, implemented 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 3D 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 sales set at more \$9.9 billion and topping \$16.2 billion in 2010. This highly-profitable situation is playing itself out throughout the world. A report released by Niko Partners (a Shanghai-based market researcher) predicted China's online game revenue to reach \$5.8 billion for 2011 – an sector expected to grow by an annual rate of 33.5 percent. According to the ESA (Entertainment Software Association) more than 60% of Americans aged 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, constantly improving graphics appears to be 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 (29 April 2008) and grossing more than \$500 million in its first week. In less than a week, the game had sold over 10 million copies (Ortutay, 2008).

This ever constant push for “immersive and more realistic” computer games has resulted in a significant number of 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, as with other special effects, 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, reflections and specularities. 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 rendering algorithms 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. 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. Following the iPhone SDK release, there has been an enormous interest in creating applications and especially games targeting this platform. This interest has resulted in more than 500 applications (with 241 in the game category) being available on launch date of Apple's delivery platform, the AppStore. Games targeting this platform have an even harder time when it comes to performance balancing. For example, early iPhones featured a 620 MHz ARM 1176 CPU underclocked to 412 MHz with its Graphical Processing Unit (offering support for OpenGL ES) being a PowerVR MBX Lite 3D unit (Apple, 2008). Even though the 3D capabilities of these devices have been improved since 2008, running high-quality immersive games with PC-like special effects on the iPhone remains a problem and is a classic example of the need for performance balancing, especially when rendering shadows and other advanced special effects.

General-purpose computing on graphics processing units is the parallel computing technique of using a Graphic Processor Unit (GPU), which typically handles computer graphics computations, in conjunction with a CPU to perform computations traditionally handled solely by the CPU. Using these specialised graphics processors as "mini CPUs" is the direct result of the programmable GPU evolving into a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and very high memory bandwidth (NVIDIA, 2009). Modern-day programmable GPUs are thus especially well-suited to address problems that can be expressed as data-parallel computations with a high ratio of arithmetic operations to memory operations. Parallel

computing has also become commonplace with technologies like AMD's HyperTransport enabling high-performance reconfigurable computing and general-purpose computing on graphics processing units (GPGPU) allowing for highly parallel, multithreaded, many-core processing.

Recent work into the utilisation of GPUs for General Parallel Computations (NVIDIA's CUDA, for example) ranges from CPU-GPU communication management (Jablin et al, 2011), multi-GPU and multi-CPU parallelisation for physics simulations (Hermann et al, 2011), the development of physics engines featuring automatic CPU-GPU process distribution (Joselli et al, 2008), adaptive game loop architectures with CPU-GPU task distribution (Joselli et al, 2009), the modelling of GPU-CPU workloads for General Parallel Computations (Kerr et al, 2010), the acceleration of graphics applications through the implementation of GPU/CPU caches (Likun and Dingfang, 2008), NVIDIA GPU and ARM CPU integration (Moore, 2011), the parallel processing of matrix multiplication in CPU and GPU environments (Ohshima et al, 2006), the concept of scalable heterogeneous computing (Nickolls and Dally, 2010), the optimisation of data parallel execution on GPUs (Perumalla, 2008), CPU-GPU parallel optimisations for SIMD/SPMD computing (Qi Ren, 2011), a number of proposed GPU-CPU communication models (Shainer et al, 2011), the use of asynchronous stencil kernels for hybrid CPU/GPU systems (Venkatasubramanian and Vudac, 2009), the task scheduling of parallel processes in a collaborative CPU-GPU environment, the deployment of CPU and GPU-based genetic algorithms on heterogeneous devices (Wilson and Banzhaf, 2009), a performance study on GPU/CPU resource interference (Yamagiwa and Wada, 2009), the execution of database applications using GPGPU programming (Zidan et al, 2011) to an architectural proposal for hybrid GPU/CPU middleware solutions (Zink, 2008).

Research into parallel or distributed rendering has also been conducted since the early 1980s (Crockett, 1995) with Silicon Graphics Inc, for example, originally defining OpenGL as a client-server API (Fosner, 1996). What hasn't picked up great momentum is the utilisation of the CPU in an attempt to free up GPU resources and, in turn, to accelerate graphics performance. Research has mostly been limited to GPU-exclusive parallel rendering environments such as render farms, graphic clusters and visual simulation applications where multiple display systems are interconnected and rendered to concurrently (Fangerau et al, 2010). The scene, subdivided into a sequence of frames, is thus distributed amongst these interconnected display systems, resulting in significantly faster rendering times (Allard and Raffin, 2005). In another example of multi-GPU rendering, Isard et al (2002) proposes a system for the distributed rendering of soft shadows.

Adapting this approach for real-time, interactive graphics as found in modern DirectX and OpenGL-based computer games to date entails distributing the rendering task across several interconnected GPUs (via a technology such as NVIDIA's Scalable Link

Interface – a multi-GPU approach for linking two or more video cards together to produce a single output). However, this thesis proposes the unification of the parallel compute engine present in modern GPUs with that of multi-core CPUs to allow for the rendering of complex geometric environments without the overburdening of scarce computational resources.

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). This constant demand is typically met by more expensive/better hardware which, in turn, results in an even higher need for realism and performance. One possible consequence of advanced hardware such as NVIDIA's GeForce 500 Series is that the GPU is often fully utilised while the CPU, by comparison, sits relatively idle (especially the case with modern multi-core CPUs) – underutilisation of the GPU is a conjecture that drove the thesis. Part of the purpose of the study was precisely to test whether the CPU was sufficiently underutilised to allow for increased use; if so, then the CPU can be considered a less than fully utilised processing resource with the GPU being a relatively over-utilised one. Another consequence is the global use of unnecessarily sophisticated rendering algorithms providing a quality of detail that is inappropriate for a given context – for instance, highly accurate shadows for very distant objects (the system to be discussed in this thesis will, in contrast, render the shadows of near objects via stencil shadow volumes and distant objects via blob shadows).

Purpose of the Study

The primary purpose of this study is to examine the overall quality and performance impact resulting from the global use of unnecessarily sophisticated rendering algorithms and, secondarily, to gauge the extent of GPU over-utilisation and CPU under-utilisation. Then, based on these findings, the study examines whether improved rendering quality and performance can be achieved through appropriate algorithm selection both within a given scene and in successive scenes and, as a proof of concept approach, through load-balancing between the CPU and GPU.

The overarching agenda is to explore a new paradigm for game development that will be less resource hungry but nevertheless not have a net-negative impact on rendering quality, thereby facilitating the development of games that fully utilise all processing power at hand. The hope is that the paradigm will be applicable both in the context of highly polished, GPU-hungry PC titles and in the context of mobile games, thus

forestalling a situation where, for example, an iPhone's PowerVR SGX GPU is fully utilised while its 800 MHz ARM CPU sits relatively idle.

Performing the Study

The study is performed through the implementation of a wide and representative range of rendering and physics algorithms (organised into performance-impacting groups). A platform supporting the swapping out of rendering algorithms and physics calculations as well as the selective transfer of tasks between the CPU/GPU is built. This platform enables the detailed benchmarking of the various implemented algorithms which, in turn, allows for the definition of a fuzzy-logic based expert system and real-time rendering engine. Using this benchmarked performance data, the rendering engine and fuzzy-logic based selection engine analyse the 3D environment being rendered to determine the best solution to a given problem and, as proof of concept, to combine the parallel compute engine in modern GPUs with that of multi-core CPUs. This allows for the rendering of complex geometric environments through the real-time swapping of rendering algorithms and the rendering of reflections and physics computations through the effective distribution of processing tasks between the CPU and GPU.

Scope

This study is inspired by earlier work on shadow rendering – please see the MSc dissertation, *An Empirically Derived System for High-Speed Shadow Rendering* (Rautenbach, 2008). In that case, several shadow algorithms were benchmarked and analysed, specifically: 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 own algorithm based on shadow volumes, spatial subdivision and instruction set utilisation. This critical analysis allowed us to assess the relationship between shadow rendering quality and performance. It also allowed for the isolation of key algorithmic weaknesses and possible bottleneck areas. Focusing on these bottleneck areas, several possibilities of improving the performance and quality of shadow rendering, both on a hardware and software level, were investigated. Primary performance benefits were 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 were achieved by combining the depth-fail stencil shadow volume algorithm with dynamic spatial subdivision. Using the performance data gathered during the analysis of various shadow rendering algorithms, the system was able to dynamically swap out shadow rendering algorithms based on environmental conditions.

Our dynamically scalable interactive rendering engine as presented in this thesis features not only dynamic shadow algorithm swapping but also the dynamic swapping and, in the case of environmental mapping, CPU/GPU allocation of shaders, local illumination configurations, a number of reflection and refraction implementations and approaches, physics calculations, particle effect calculations and numerous post-processing effects.

Implemented shader and related lighting effects include: simple light mapping, basic directional lighting, normal mapping, specular highlights, volumetric fog, a detailed lighting model, ambient occlusion, High Dynamic Range Lighting and parallax mapping.

Local illumination approaches include the limiting of the number of light sources in an attempt to reduce GPU utilisation and the lifting of this limitation while occluding local light sources (a technique used to approximate the effect of environment lighting as an attempt to simulate the way light radiates in real life). This implementation was also extended with the inclusion of HDR lighting.

Reflection and refraction implementations and approaches include: basic environmental mapping, CPU-based cube mapping, refractive environmental mapping and the extension of these reflection and refraction algorithms through the addition of the Fresnel effect and chromatic dispersion.

Physics calculations include: acceleration, force, linear momentum, gravitational pull, projectile simulation through trajectory paths, friction and collision detection. A physics-based particle generator is also included.

Post-processing shader implementations and related lighting approaches include: displacement mapping, bloom effects, ambient occlusion, depth of field and halo effects.

Implemented Algorithms

The presented rendering engine utilises a number of base rendering algorithms commonly implemented in high-end rendering engines such as id Tech 5 (id Software, 2011), Blizzard's StarCraft II Engine (Blizzard, 2010) and Epic Games' Unreal Engine Technology (Epic, 2012). These algorithms make up the core of all current generation 3D games (such as id Software's *Rage*) with future technologies such as id Tech 6 – an upcoming game engine under preliminary development by id Software – aiming for a mixed environment where ray tracing and classic raster graphics are to be merged. That said, as stated by id Software's technical director, John Carmack, id Tech 6 will utilise hardware that "doesn't exist right now" (Carmack, 2011). The presented

rendering algorithms and approaches were thus selected as they are utilised, in various combinations, by a majority of high-end 3D titles. Examples include *Rage* (released 2011), *Grand Theft Auto IV* (released 2008), *Hitman: Absolution* (to be released 2012), etc.) – either in their basic/core form or as a variation/extension of the original. The presented algorithms also cover the entire realm of interactive rendering as found in modern 3D games – shadow rendering, local illumination, reflection and refraction, physics calculations, particle effects and numerous post-processing special effects.

The graphics academic research community is, of course, constantly researching new rendering algorithms, sometimes improving on efficiency, sometimes on realism, and sometimes on both. In principle, some of these algorithms might have been included in our experiments. However, since the aim of the presented rendering engine is to serve as a proof of concept, it was decided to limit the scope of the project to established core algorithms already in widespread use. In future experimentation, the implemented algorithms can easily be extended or replaced. More algorithms could, for example, be benchmarked and added to our selection engine's knowledge base. The implemented rendering engine is also highly expandable and alternate rendering solutions, whether GPU or CPU based, can be implemented and loaded into the engine as additional dynamic link libraries. Alternate algorithmic performance improvements can also be pursued.

Furthermore, the presented algorithms and rendering approaches utilise the power of current generation GPUs and graphics APIs to the fullest. For example, the bump mapping and displacement mapping approaches presented in this thesis leverage the hardware tessellation engine provided by Microsoft's latest API, DirectX 11, as well as today's high-end GPUs to generate more triangles from existing geometry. The result of this is extremely high-resolution displacement and bump maps that appear truly 3D. The downside to this advancement and realism is, unfortunately, a decrease in rendering performance. Since hardware resources are limited, these advancements are much better utilised when implemented for close-up, important objects with distant objects being tessellated to a lesser degree – one result of the presented selection engine.

Another modern rendering technique implemented by the proposed system is ambient occlusion. This technique, as a way to enhance the ambient light term such that shadows and light emission from local features are included, was thoroughly investigated by Langer and Buelthoff (2000), but only recently, with the release of DirectX 10 and thanks to the efforts of Landis, McGaugh and Koch, who in 2010 received the Scientific and Technical Academy Award for their work on ambient occlusion at Industrial Light & Magic, started to appear in real-time rendering applications. This technique, as a base approach, can be found in many newer games, with high-definition ambient occlusion (HDAO) and horizon-based ambient occlusion (HBAO) being implemented as variations. The presented study's algorithms, as mentioned, were thus selected because they are being utilised throughout the industry

and because they are often extended and varied as required. These extensions, whether in respect of shadow rendering, ambient occlusion or transparent anti-aliasing, nevertheless remain variations on the original. Ambient occlusion as a post-processing special effect, for instance, first appeared with DirectX 9-generation games and has steadily increased with the release of DirectX 10 and now, DirectX 11. These effects are primarily used for added realism and image quality.

Specifically, the algorithms implemented and benchmarked as part of this thesis (all, in various combinations and in some degree or another, utilised by modern 3D titles as either basic/core rendering algorithms and/or as post-processing special effects) include:

Shadows	Shader & Lighting Effects	Reflection and Refraction	Physics	Post-processing shaders
Stencil shadow volumes	Simple light mapping	Basic environmental mapping	Newtonian physics	Displacement mapping
Hardware shadow mapping	Basic directional lighting	CPU-based cube mapping	Physics-based particle effects	Bloom effects
McCool's shadow volumes	Normal mapping	Refractive environmental mapping		Ambient occlusion
Chan and Durand's hybrid algorithm	Specular highlights	Fresnel effect & chromatic dispersion		Depth of field
Thakur et al's algorithm	Volumetric fog			Halo effects
Rautenbach et al's spatial subdivision algorithm	Ambient occlusion			
	High Dynamic Range Lighting			
	Parallax mapping			

Table 1.1 Primary algorithms/rendering approaches implemented and benchmarked.

The algorithms listed in Table 1.1 were accordingly referenced (please see the bibliography, pg 179 – 199). For example, stencil shadow volumes, as an established core algorithm already in widespread use, was first proposed by Crow in 1977 with the first commercial application as a real-time shadowing technique being the release of id Software's Doom 3 (2004/5). The development and evolution of this algorithm, through use of the stencil buffer, are thoroughly discussed and referenced throughout. Subsequent approaches, such as Thakur et al.'s shadow generation using a discretized shadow volume in angular coordinates (2003), Chan and Durand's hybrid approach (2004) and Rautenbach et al.'s spatial subdivision approach (2008), as the most recent improvements on the original, were also discussed. As previously mentioned, since it was decided to limit the scope of the project to established core algorithms already in

widespread use, all the other algorithms were dealt with in a similar manner, with both historic and recent material being referenced – Table 1.2 gives a summary of this.

Shadows	Shaders, Lighting & Reflection/Refraction	CPU/GPU, Hybrid Rendering, Tech	Physics, AI
<ul style="list-style-type: none"> ▪ Akenine-Möller T. and Assarsson U. (2002) ▪ Atherton P., Weiler K. and Greenberg D. (1978) ▪ Bergeron, P. (1985) ▪ Blinn J. (1988) ▪ Bouknight W. and Kelly K. (1970) ▪ Brabec S. and Seidel H. (2002) ▪ Brotman L.S. and Badler N.I. (1984) ▪ Carmack J. (2000) ▪ Chan E. and Durand F. (2004) ▪ Crow F. (1977) ▪ Dimitrov R. (2007) ▪ Drettakis G. and Fiume E. (1994) ▪ Everitt C., Rege A. and Cebenoyan C. (2001) ▪ Everitt C. and Kilgard M. (2002) ▪ Fernando R., Fernandez S., Bala K. and Greenberg D. (2001) ▪ Haines E. (2001) ▪ Heidmann T. (1991) ▪ Heidrich W., Brabec S. and Seidel H. (2000) ▪ Hourcade J.-C. and Nicolas A. (1985) ▪ Isard M., Shand M., and Heirich A. (2002) ▪ Kersten D., Mamassian P. and Knill D. (1994) ▪ Kersten D., Mamassian P. and Knill D. (1997) 	<ul style="list-style-type: none"> ▪ Alard J. and Raffin B. (2005) ▪ Angel E. (2006) ▪ AMD (2011) ▪ Bier E. and Sloan K. (1986) ▪ Blinn J. (1977) ▪ Blinn J. and Newell M. (1976) ▪ Bouknight W. and Kelly K. (1970) ▪ Boulanger K., Pattanaik S. and Bouatouch K. (2006) ▪ Cabral B., Max N. and Springmeyer R. (1987) ▪ Cohen et al. (1998) ▪ Crytek 2 (2011) ▪ Drettakis G. and Fiume E. (1994) ▪ Gray K. (2003) ▪ Goral C, Torrance D., Greenberg D. and Battaille B. (1984) ▪ Greene N. (1986) ▪ Heckbert P. (1986) ▪ Hearn D. and Baker M. (2004) ▪ Kalogirou, H. (2006) ▪ Landis, McGaugh and Koch (2010) ▪ Langer, Bühlhoff (2000) ▪ Levoy M. and Hanrahan P. (1996) ▪ Microsoft (2006 – 2011) ▪ Mikkelsen M. (2008) ▪ Nguyen H. (2007) ▪ NVIDIA (2009-2011) 	<ul style="list-style-type: none"> ▪ August D., Huang J., Jablin T., Kim H., Mason T., Prabhu P., Raman A. and Zhang Y (2011) ▪ Epic Games (2012) ▪ Harbour J.S. (2004) ▪ Hermann E., Raffin B., Faure F., Gautier T., Allard J. (2011) ▪ Fangerau J., Krömker S. (2010) ▪ Fernando R. (2004) ▪ Future Chips (2011) ▪ Huang J., Raman A., Zhang Y., Jablin T., Hung T., and August D. (2010) ▪ Id Software. (2011) ▪ Intel. (2011) ▪ Jablin T., Prabhu P., Jablin J., Johnson N., Beard S., August D. (2011) ▪ Jablin T., Jablin J., Prabhu P., Liu F, and August D. (2012) ▪ Joselli M., Zamith M., Clua E., Montenegro A., Leal-Toledo R., Conci A., Pagliosa P., Valente L., Feijó B. (2009) ▪ Moore S. (2011) ▪ Nickolls J., Kirk D. (2009) 	<ul style="list-style-type: none"> ▪ Belleman R., Bedorf J., Zwart S. (2008) ▪ Choppin B. (2004) ▪ Crossno P. and Angel E. (1997) ▪ Flynt J. and Salem O. (2004) ▪ Funge J. (1999) ▪ Giarratano J., Riley G. (2005) ▪ Hahn J. (1988) ▪ Halliday D., Resnick R. and Walker J. (2007) ▪ Hecker C. (2000) ▪ Hermann E., Raffin B., Faure F., Gautier T., Allard J. (2011) ▪ Hubbard P. (1996) ▪ Ignizio J. (1991) ▪ Joselli M., Clua E., Montenegro A., Conci A., Pagliosa P. (2008) ▪ Lay D. (2005) ▪ Mamdani E. H., Assilian S. (1975) ▪ Moore M. and Wilhelms J.

<ul style="list-style-type: none"> ▪ Kilgard M. J. (1999) ▪ Kirsch F. and Doellner J. (2003) ▪ Kolic I., Mihajlovic Z., Budin L. (2004) ▪ Lauritzen A. (2006) ▪ Lokovic T. and Veach E. (2000) ▪ McCool M. D. (2000) ▪ Nishita T. and Nakamae E. (1985) ▪ Rautenbach, P. (2008) ▪ Rautenbach P., Pieterse V., Kourie D., (2008) ▪ Reeves W., Salesin D. and Cook R. (1987) ▪ Segal M., Korobkin C., van Widenfelt R., Foran J. and Haeberli P. (1992) ▪ Thakur K., Cheng F. and Miura K.T. (2003) ▪ Williams L. (1978) ▪ Woo A., Poulin P. and Fournier A. (1990) 	<ul style="list-style-type: none"> ▪ Peercy M., Airey J. and Cabral B. (1997) ▪ Pharr M., Fernando R. (2005) ▪ Phong B. (1975) ▪ Piegl L. (1993) ▪ Policarpo F., Oliveira M. (2006) ▪ Fernando R. (2004) ▪ Segal M., Korobkin C., van Widenfelt R., Foran J. and Haeberli P. (1992) ▪ Sillion F., Puech C. (1989) ▪ Torrance K. and Sparrow E. (1967) ▪ Wagner F., Schmuki R., Wagner T. and Wolstenholme P. (2006) ▪ Warren J. and Schaefer S. (2004) ▪ Warn D. (1983) ▪ Wenzel C. (2006) ▪ Wloka M. (2002) ▪ Yamagiwa S., Wada K. (2009) 	<ul style="list-style-type: none"> ▪ Nguyen H. (2007) ▪ NVIDIA (2009-2011) ▪ Ohshima S., Kise K., Katagiri T., Yuba T. (2006) ▪ Pajot A., Barthe L., Paulin M. and Poulin P. (2011) ▪ Pharr M. and Fernando R. (2005) ▪ Rabin S. (ed.) (2005) ▪ Qi Ren, D. (2011) ▪ Shainer G., Lui P., Liu T. (2011) ▪ Venkatasubramanian S., Vudac R. (2009) ▪ Wilson G., Banzhaf W. (2009) ▪ Yamagiwa S., Wada K. (2009) ▪ Zidan M., Bonny T., Salama K. (2011) ▪ Zink B. (2008) 	<ul style="list-style-type: none"> (1988) ▪ Nickolls J., Dally W. (2010) ▪ Nilsson J. (1986) ▪ Reeves W. (1983) ▪ Reeves W. and Blau R. (1985) ▪ Reynolds C. (1987) ▪ Salton G. (1987) ▪ Watt A. and Watt M. (1992) ▪ Witkin A. and Heckbert P. (1994)
---	---	--	---

Table 1.2 References of relevant algorithms and approaches in widespread use and utilised by the proof of concept rendering engine.

The Selection Engine and CPU-GPU Process Allocation

The presented study analyses a large number of rendering algorithms and approaches with the aim of highlighting the need for a system to primarily control the real-time selection and, as a secondary aim, CPU/GPU-process allocation of rendering algorithms and special effects groupings based on environmental conditions. We present such a solution through the critical analysis of numerous real-time rendering algorithms and the construction of an empirically derived system for high-speed rendering. This critical analysis allows us to assess the relationship between rendering quality and performance.

Using the gathered performance data, we are able to define a fuzzy logic-based selection engine to control the real-time allocation and selection of rendering algorithms based on environmental conditions. This system ensures the following: nearby effects

are always of high-quality (where computational resources are available), distant effects are, under certain conditions, rendered at a lower quality and the frames per second rendering performance is always maximised.

The CPU-GPU process allocation sub-system is used to control performance and quality and serves chiefly as proof of concept. It is only used for CPU-based cube mapping (the real-time allocation of the presented cube mapping approach), PhysX-based physics calculations and the execution of the presented particle system (illustrating that the CPU can, in practice and under significant load, be used to free up valuable GPU resources). It is also shown that the selection engine can be extended to facilitate CPU-GPU process allocation. This approach is similar to the work done by Pajot et al (2011) in which bi-directional path-tracing was divided into a number of parallel processes executed on both the CPU and GPU. Their approach resulted in a performance gain of more than ten times that of other bidirectional path-tracing implementations. Larger scale research in the field of hybrid rendering is also being done by Intel (2011) who is currently developing a hybrid rendering and visualisation system to combine the strengths of different rendering algorithms, hardware models and display technologies while avoiding their weaknesses. Similarly, Bernhardt et al (2011) presents a system for real-time terrain modelling via CPU-GPU coupled computation – a system efficient and fast enough to display terrain morphing in real time. In contrast to the forgoing research that distributes the algorithmic logic over the respective processors (i.e. over the CPU and GPU), our utilisation of the CPU is as an *alternative* computational resource to the GPU when the latter is under high load.

Hence, since an all encompassing production system would have required the implementation of both a CPU and GPU-version of the majority of presented algorithms and/or rendering approaches, it was decided to limit the presented GPU-CPU process allocation approach to cube mapping and physics processing. An alternative approach initially investigated was the implementation of a generic CPU-based rendering library. However, given the sheer amount of work involved in addition to the development of a fully-functional, DirectX 10-based rendering engine, it was decided that CPU vs. GPU-cube mapping and CPU-based physics processing would serve as evidence of such a system's inherent benefits both in the realm of high-quality rendering and general computations. As an aside, with reflections it was found that, when the GPU is fully utilised and when additional computational resources are required, and if the CPU is not fully utilised and can be utilised to lighten the GPU load, then performance gains are achieved by switching to CPU-based cube mapping. Physics processing showed similar results.

Broad Findings

The presented study provides *prima facie* evidence that the process of dynamically cycling through the most appropriate algorithms based on ever-changing environmental conditions (along with the unification of the CPU and GPU, as secondary objective) allows for maximised rendering quality and frame-rate performance and shows that it is possible to render high-quality visual effects without overburdening scarce computational resources.

Immersive rendering approaches used in conjunction with AI subsystems, game networking and logic, physics processing and other post-processing special effects are extremely processor intensive and can often only be implemented on high-end hardware. Only by cycling algorithms and distributing computations based on environmental conditions can high-quality real-time special effects find application in non-traditional gaming devices such as tablet PCs and smart phones. Also, as mentioned, using this system ensures that performance vs. rendering quality is always optimised, not only for the game as a whole but also for the current scene being rendered. Some scenes might, for example, require more computational power than others, resulting in noticeable slowdowns under the conventional processing paradigm, but slowdowns not experienced in the proposed new paradigm, thanks to the presented system's dynamic cycling of rendering algorithms and its unification of the CPU and GPU for cube mapping and physics processing.

1.3 Dissertation Structure

To explain the work that has been done to investigate the feasibility of the new gaming paradigm that has been proposed, this thesis has been partitioned into 2 parts.

The first part consists of the first two chapters in which introductory background information to the study is given, as well as an overview of the software framework system that was put in place to carry out the study. The second part shows how empirical investigations provided information that was subsequently used to drive a real-time rendering engine, built in terms of the new games processing paradigm.

Thus, in Part I, this current chapter presented an historical account of the general research domain, research problem and overall dissertation structure.

Also in Part I, Chapter 2 presents the general design and implementation of a generic game engine (the core of our dynamically scalable interactive rendering engine/benchmarking environment). This base implementation is subsequently extended into an all-encompassing solution for the rendering of computationally intensive 3D environments through the addition of several rendering algorithms and techniques,

specifically: shaders, local illumination, reflection and refraction, HDR lighting, shadows, physics, particles and post-processing special effects (Chapter 3).

The second part of the thesis consist of three chapters, with the first of these, Chapter 4, presenting the critical analysis and detailed benchmarking of the previously discussed rendering techniques. The knowledge base of our selection engine draws heavily on these experimental results. Each of the presented rendering techniques are categorised based on the level-of-detail/rendering quality and the associated computational impact.

Following this, Chapter 5 discusses the previously mentioned selection engine in much more detail. It also presents the critical analysis of our empirically derived system. 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 rendering algorithms and approaches with the aim of highlighting the need for a system to control the real-time selection and CPU/GPU-process allocation (as proof of concept) of rendering algorithms and special effects groupings based on environmental conditions. We present such a solution through the critical analysis of numerous real-time rendering algorithms and the construction of an empirically derived system for high-speed rendering. This critical analysis allows us to assess the relationship between rendering quality and performance.

Using the gathered performance data, we are able to define a fuzzy logic-based selection engine to control the real-time selection (and to a limited degree, the allocation) of rendering algorithms based on environmental conditions. This system ensures the following: nearby effects are always of high-quality (where computational resources are available), distant effects are, under certain conditions, rendered at a lower quality and the frames per second rendering performance is always maximised.

An abstract model illustrating the generality of the proposed system is given in Figure 1.9. This figure shows the fuzzy logic-based selection engine, the Direct3D-based rendering engine and the rendering algorithms selectable based on environmental conditions. The selection engine (Chapter 5) shown here controls, as mentioned, the selection and allocation of algorithms by correlating the properties of the scene being rendered with obtained algorithmic performance data. The core implementation of the rendering engine (Chapters 2 and 3) subsequently serves as a scalable interactive testing environment and is an adequate platform for the purposes of this thesis, in which the objective is to experiment with the impact of various algorithms when rendering computationally intensive 3D environments – specifically, as shown in Figure 1.9, the

“rendering module” deals with the actual Direct3D API calls and scene geometry with the “level initialisation module” being tasked with the loading of octree-based “maps” or “scenes”. The “physics module”, in turn, controls basic world dynamics, i.e. whether the “player” can walk through walls or not, whether a specific medium is solid (such as a floor) or liquid (such as water) and how the “player”, controlled via the “input module”, interacts with these materials.

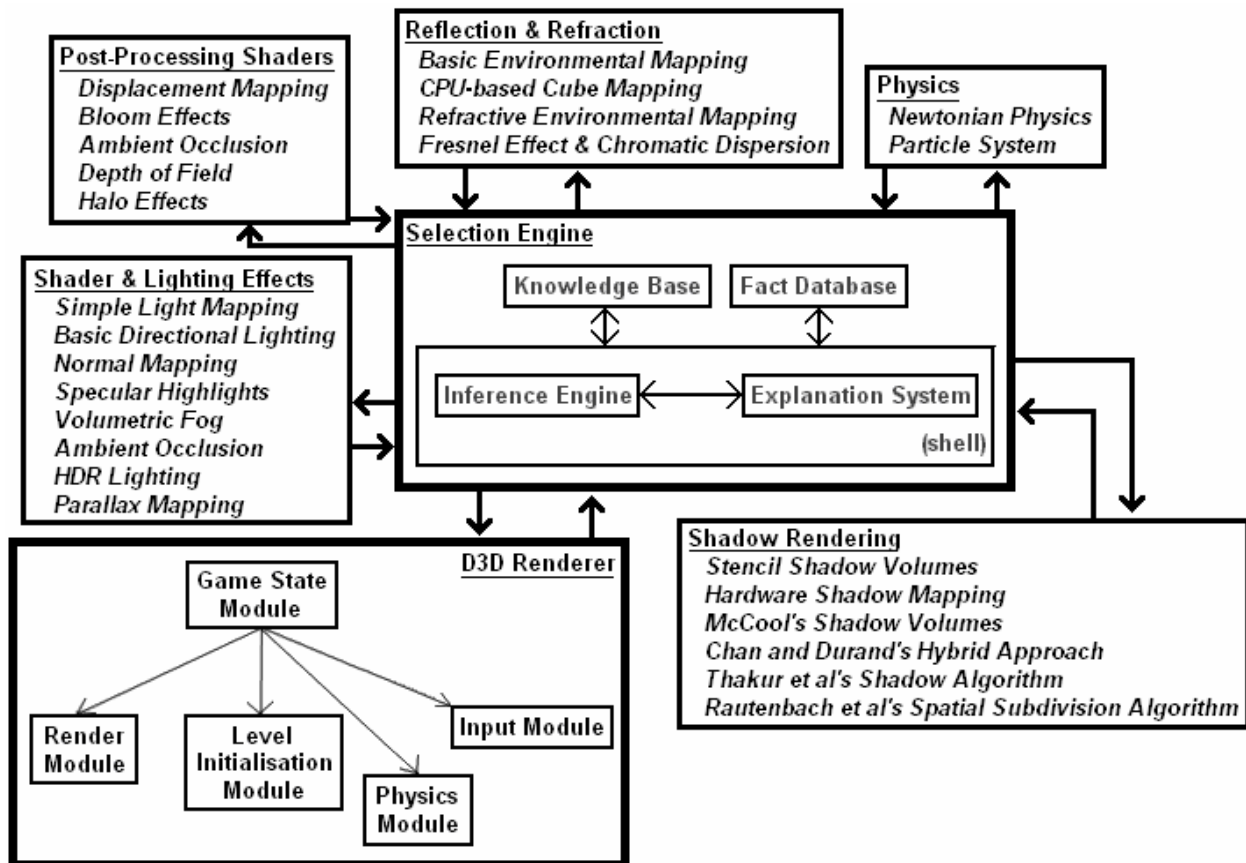


Figure 1.9 An abstract model illustrating the generality of the proposed system.

Please note that unless otherwise stated, that all screenshots and/or illustrative images have been rendered using our dynamically scalable interactive rendering engine. The accompanying CD contains implementation source code and several videos (including a high-definition video showcasing the rendering engine).

Creating an Interactive 3D Environment

Chapter 2 starts by outlining the general design of a generic game engine (with the aim of providing background information on 3D engine design). Focus then shifts to the implementation of a basic DirectX 10 3D interactive environment featuring mesh-loading, texture mapping, movable light sources, a GUI and stencil shadow volumes (as this study is conducted through the implementation of such a system).

Outline:

- Game engine architecture
- Game initialisation and shutdown
- The game loop
- Creating a basic interactive DirectX 10 3D environment

2.1 Game Engine Architecture

A game engine is the central unit of any computer game and it can be described as a collection of technologies such as a sound engine, AI subsystem, physics engine, networking subsystem, 3D renderer, input control system, etc. The number of subsystems provided is highly dependant on the developer's requirements and the implementation platform of choice.

Game engines, built upon various APIs such as DirectX and OpenGL, are normally designed with software componentry in mind. This allows for decomposition of the engine, resulting in numerous functional units. By designing component-based engines, we are able to replace provided technologies with other third-party or in-house developed units as needed. For example, a game engine's renderer, physics engine or sound system can easily be replaced by an improved or alternate version in a plug-and-play fashion.

The term "game engine" has existed for some time now, but only became truly common in the mid-1990s when developers started licensing the core code of other games for their own titles. This reuse led to the development of high-end commercial game engines and middleware providing game developers with a number of game creation tools and technical components – i.e. accelerating the game development process. The following list gives some idea of what might be supported by a commercially targeted game engine:

1. 3D Engine
 - Direct3D 10 renderer for Microsoft Windows based systems
 - OpenGL renderer for MacOS X, Linux, Unix, etc
 - High Level Shading Language (HLSL) and C for Graphics (Cg) shader support
 - Normal mapping
 - Environmental mapping
 - Displacement mapping
 - High Dynamic Range lighting
 - Depth-of-field
 - Motion blur
 - Bloom and sobel effects (for older hardware support)
 - Rome algorithmic based Level Of Detail automatic adaptation system
 - Dynamic lighting and shadowing
 - Soft shadows
 - Specular reflections with specular bump maps
 - Reflective water (with refraction)
 - Highly efficient occlusion culling
 - Dynamically deformable and destroyable geometry



- Cg rendered moving grass, trees, fur, hair, etc
 - Advanced Particle System: model and sprite based (snow, smoke, sparks, rain, ice storms, fire storms, volumetric clouds, weather system, etc)
 - Non-Player Character (NPC) Material Interaction System (vehicle sliding on ice, etc)
2. Artificial Intelligence (AI) Subsystem
 - Cognitive model based NPC AI (no way-point system)
 - Intelligent non-combat and combat NPC interaction
 - Conversation system
 - NPCs make decision to fight, dodge, flee, hide, burrow, etc based on player resistance
 - NPCs fall back to regroup if resistance is overwhelming
 3. Sound Engine
 - Stereo, 5.1 surround sound, quadraphonic sound, 3D spatialisation
 - Ogg (the open audio container format) and adaptive differential pulse-code modulation (ADPCM) decompression
 - Real-time audio file stitching (Ogg and Wave)
 - Distant variant distortion
 - Material based distortion (e.g. under water distortion of helicopter hovering overhead)
 - Environmental DSP (Digital Signal Processing)
 4. Physics Engine
 - Realistic object interaction based on Newton's Laws
 - Particle system inherits from Physics Engine
 - NPCs interact with objects realistically
 - All objects react based on force exerted and environmental resistance
 5. Networking System
 - Up to 64-player LAN and 32-player internet support
 - High-latency, high-packet loss optimisations
 - Predictive collision detection performance enhancement
 6. Development
 - In-game level and terrain editor
 - Exporters (meshes, brushes, etc)
 - C++ written code compiled to modular design
 - Event debugger and monitoring tools built into engine
 - Shader editor

Creating a game engine supporting all the above listed elements takes a lot of time, money, skilled developers and support infrastructure. However, most of the listed features can be added to an engine in a pluggable fashion. Hence, designing and implementing a basic first-person shooter game engine can be done by one programmer, time being the only limit in regard to the number of supported features. It is

thus of critical importance to have a well-defined architecture, without which the source code of an engine would not be extendible, maintainable or easily understandable.

The source code of a game can be divided into two units, namely, the game-engine code and the game-specific code. The *game-specific code* deals exclusively with in-game play elements, for instance, the behaviour of non-player characters, mission-based events and logic, the main menu, etc. Game-specific code is not intended for future re-use and thus excluded from the game engine code. *Game-engine* code forms the core of the entire game implementation with the game-specific code being executed on top of it. The game engine is separate from the game being developed in the sense that it provides all the technological components without any hard coded information about the actual gameplay. Game-specific and engine-specific code are commonly compiled to dynamic-link libraries for easy distribution, modification and updating.

Game-engine code and game-specific code can be designed and integrated using one of the following architectures: ad-hoc, modular or the directed acyclic graph architecture (DAG).

Ad-hoc architecture describes a code base developed without any specific direction or logical organisation (Eberly, 2001). For example, a developer simply adds features to a game engine on an “as-needed” basis. This form of code organisation leads to very tight coupling (a high level of dependency) between the game-specific and game-engine code – something that is acceptable in small game projects such as mobile and casual games.

Modular architecture organises the code base into modules or libraries with a module consisting of numerous functions available for use by other modules or libraries (Flynt and Salem, 2004). Using this design, we are able to add and change modules as needed. Middleware such as a third-party physics engine can also easily be integrated into a modular designed code base. Modular organisation results in moderate coupling between the various code components. However, one must take care to limit inter-module communication to avoid a situation where every module is communicating with every other module – leading to a tighter level of coupling. Figure 2.1 illustrates the modular organisation of a code base.

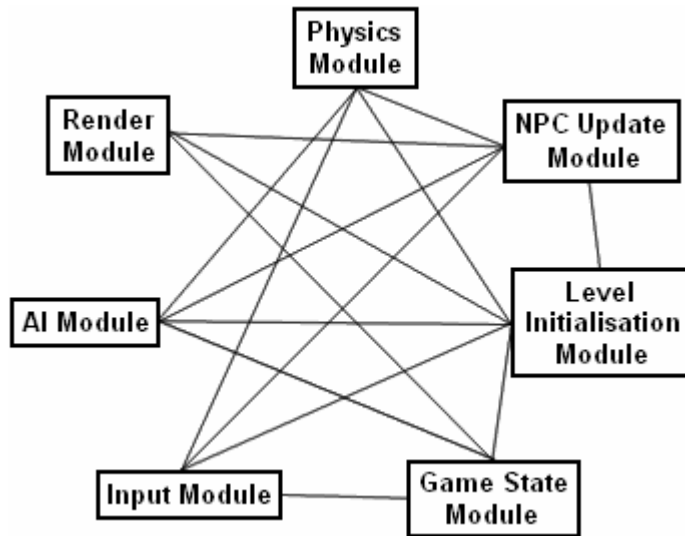


Figure 2.1 A modular architecture.

A *directed acyclic graph architecture* is a modular architecture where the inter-module dependencies are strictly regulated. A directed acyclic graph is a directed graph without any directed cycles. What this means is that for every node in the graph, there should not be any circular dependencies. For example, if the input module depicted in Figure 2.1 depends on the game state module, then the game state module cannot depend on any of the other modules that depend on the input module. The directed acyclic graph architecture is thus used to create a hierarchical design where some modules are classified on a higher level than others. This hierarchical structure, shown in Figure 2.2, ensures relative loose coupling.

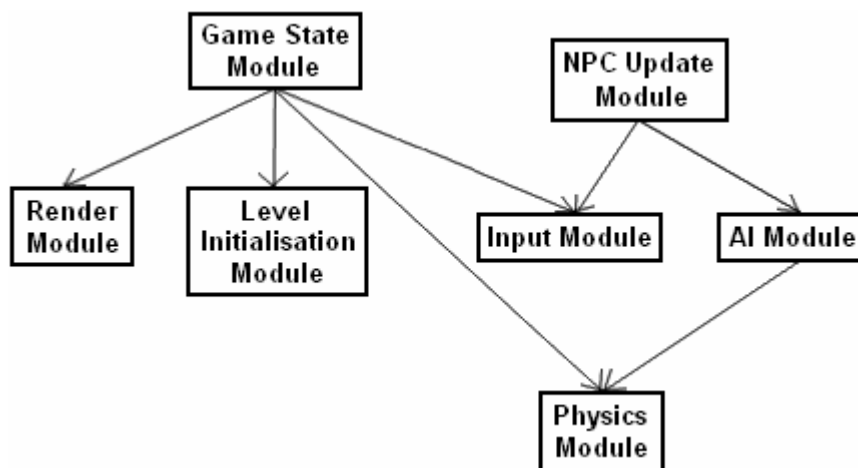


Figure 2.2 A directed acyclic graph architecture.

Other architectures also exist, each providing a different level of coupling and inter-module communication with the choice in architecture varying from application to application.

Once we have chosen the preferred overall architecture, we have to summarise all possible states our game will go through from initialisation to shutdown. Possible states (with associated events) are listed here:

1. Initialisation.
2. Enter the main game loop:
 - a. Additional initialisation and memory allocation.
 - b. Load introductory video.
 - c. Initialise and display in-game menu:
 - i. Event monitoring.
 - ii. Process user input.
 - d. Start game.
 - e. In-game loop:
 - i. Input monitoring.
 - ii. Execution of AI
 - iii. Execution of physics routines.
 - iv. Sound and music output.
 - v. Execution of game logic.
 - vi. Rendering of the scene based on the input from the user and other subsystems.
 - vii. Display synchronisation.
 - viii. Update game state.
 - f. Exit the game and return to the in-game menu.
3. Shutdown of the game if the user wishes to terminate the program.

These states will now be investigated in more detail. As mentioned, this section deals with the general design and implementation of a generic game engine which serves as the core of the proposed dynamically scalable interactive rendering engine. The next section will show how the engine allows for basic input control in the form of user-movable light sources, first-person camera and mesh. Its rendering capabilities come from the algorithms presented in Section 2.3. This extended rendering engine features dynamic algorithm swapping of shadow rendering algorithms, shaders, local illumination configurations, a number of reflection and refraction implementations and approaches, physics algorithms, a particle effect system and numerous post-processing effects. The CPU-GPU process allocation sub-system, as previously mentioned, is used to control performance and quality and serves chiefly as proof of concept. It is only used for CPU-based cube mapping (the real-time allocation of the presented cube mapping approach), PhysX-based physics calculations and the execution of the presented particle system (illustrating that the CPU can, in practice and under significant load, be used to free up valuable GPU resources).

All these implemented algorithms are presented and discussed at a source code level – a means of presentation starting below.

Please note, the question of how much information about the engine's implementation to convey in this text presented something of a dilemma. On the one hand, presenting the complete code would lead to a volume of detail would be unnecessarily overwhelming. On the other hand, it was felt that simple English narrative would not convey sufficient information about the actual depth and scope of implementation detail. For this reason, in the coming sections, the general control algorithmic structure of the implementation is explained at the source code level. It is at the reader's discretion to decide how much of the code detail to examine while reading the explanatory accompanying narrative.

2.2 Initialisation and Shutdown

The first step invoked whenever a game is executed, is initialisation. This step deals with resource and device acquisition, memory allocation, initialisation of the game's GUI, loading of art assets such as an intro video from file, etc. The first initialisation phase is commonly referred to as the front-end initialisation step to distinguish it from the level and actual game play initialisation phases. Front-end initialisation occurs prior to the game loop and is required for setting up the environment by assigning resources and loading game data and assets:

```
void FrontEndInit()
{
    AcquireResources();
    AllocMem();
    LoadAssets();
    InitGUI();
    LoadPlayerPreferences();
}
```

All devices and resources are released and final program cleanup is done during the exit state. The exit state has to release all resources and devices acquired, memory allocated and data loaded in the reverse order of the initial front-end acquisition:

```
void Cleanup()
{
    SavePlayerPreferences();
    ShutdownGUI();
    ShutdownAssetAccess();
    FreeMem();
    ReleaseResources();
}
```


It is essential to recognise the importance of error handling in the above listed initialisation and shutdown functions, especially due to the loading of files or acquisition of resources that might not exist or that might be locked by another program.

2.3 The Game Loop

The game loop allows uninterrupted execution of the game. It enables us to execute a series of tasks such as input monitoring, execution of artificial intelligence and physics routines, sound and music processing, execution of game logic, display synchronisation and so forth for every frame rendered. All these tasks are processed on a per-frame basis, thus resulting in a living world where everything happens in a seemingly concurrent manner, especially so where the computer game runs at 40 frames per second or more. A game running at 60 frames per second will result in the tasks for one frame being executed in less than 16.7 milliseconds. We will now look at the core tasks performed by a game loop.

The first task performed by any modern day game loop is timing. *Timing* allows a game to execute at a speed independent of the frame rate or processor's clock speed. Computer games developed during the 1970s and 1980s executed the maximum number of tasks possible for each frame cycle. This caused considerable variation in game speed whenever the user's hardware changed, for instance, a game running well on an Intel 80286 would be impossible to play on an Intel 80486 due to the 486's overall faster execution speed.

Each frame update reflects changes made since the previous frame and the computations performed during the game loop will be used to update all the necessary game entities accordingly. The game clock operates by using the time elapsed since the last completely executed game loop as the time measure for the current frame calculation. Timing also updates the game clock to match the actual hardware clock.

Most games released today make use of *variable frame timing*. What this means is that even though the game's frame rate may vary depending on scene complexity, the user's hardware capabilities, etc, these frame-rate changes do not affect other timing-based calculations (the game's "internal clock"). Thus, a game might operate at 60 frames per second (16.6 ms for a complete frame calculation) where the number of polygons, light sources and in-game entities are kept to a minimum. This frame rate could, on the other hand, drop to 20 frames per second (50 ms computation time) when rendering more computationally intensive scenes. The variable frame timing approach works extremely well for games targeting different platforms and hardware configurations. This is due to computations using the actual time duration of each frame as opposed to the actual frame rate.

Another key element of any game is the processing of player input. The main goal here is to minimise the amount of time taken to process an input event from the moment of occurrence up to the instant where the game can react to it – the smaller this reaction time, the more responsive the input and the greater the level of immersion. We can minimise this time by processing input at the beginning of the game loop. Networking can also be considered a form of input due to messages being received for processing.

Other tasks performed during the game loop include the execution of AI code so that NPCs can decide where to go next or what action to take, object updates, the execution of game code and scripts, the execution of physics code to ensure correct inter-object and object-entity interaction, updating the camera according to player input, animating objects and updating particle effects, etc. Collision detection (determining whether two entities have collided) and response (processing the collision and updating the health and position, for example, of related entities) is also a critical part of the game loop. Once all these tasks have successfully been executed, we can render the frame to the screen. A typical game loop looks something like this:

```
while(!ExitGame())
{
    UpdateTiming();
    InputHandling();
    UpdateNetworking();
    ExecuteScripts();
    UpdateAI();
    UpdatePhysics();
    UpdateSound();
    UpdateEntities();
    UpdateCamera();
    CollisionDetection();
    CollisionResponse();
    RenderFrame();
    UpdateGameState();
}
```

We can often improve performance by decoupling the game loop's rendering step from all the other update tasks. This will result in the rendering phase updating at a much higher rate than the other steps, however, all this will accomplish is several duplicate frames for each slower update. This situation is avoided by interpolating all the spatial values based on their previous coordinates and velocities, a process resulting in a higher frame rate. The following code sample illustrates the possible structure of a decoupled game loop:

```
while(!ExitGame())
{
    UpdateTiming();
    InputHandling();

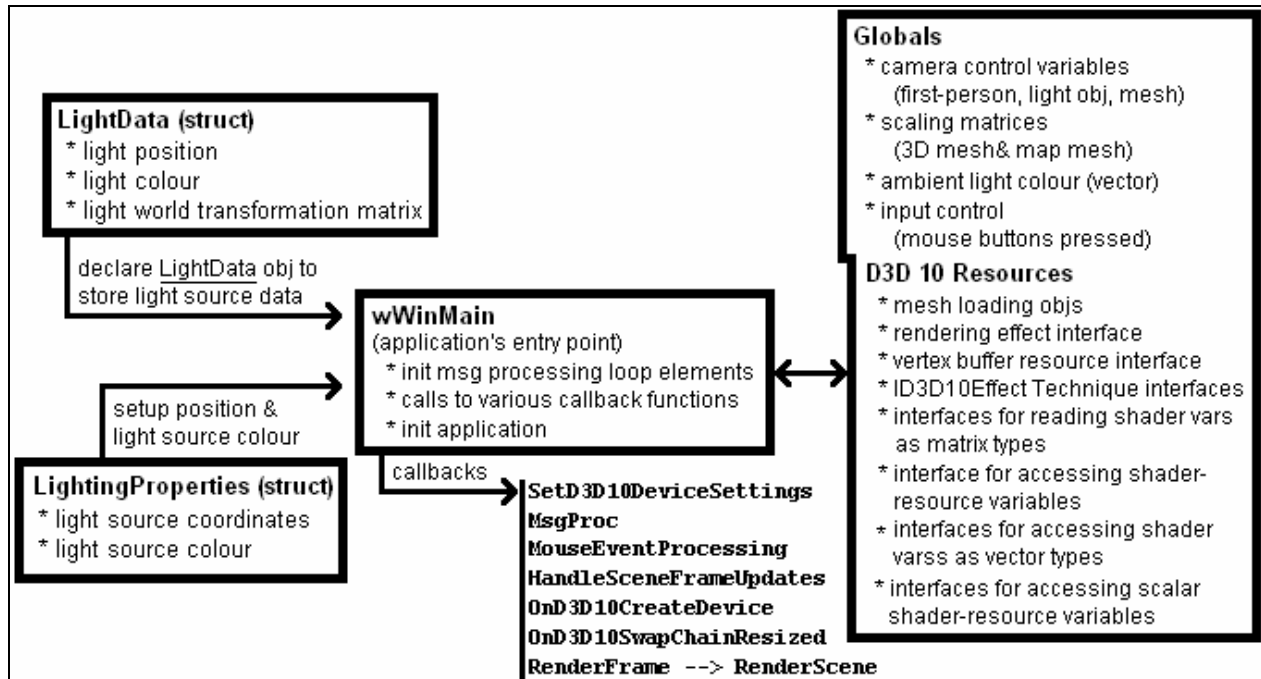
    if(UpdateWorld())
    {
        UpdateNetworking();
        ExecuteScripts();
        UpdateAI();
        UpdatePhysics();
        UpdateSound();
        UpdateEntities();
        UpdateCamera();
        CollisionDetection();
        CollisionResponse();
    }

    InterpolateObjectStates();
    RenderFrame();
    UpdateGameState();
}
```

There are also numerous other miscellaneous tasks that can be performed during the game loop. Of course tasks such as network processing are not needed for single player game modes and should be removed from the game loop to improve performance. The easiest way of doing this is to add a check at the start of the function. For example, the `UpdateNetworking` function could have a simple `if` statement returning '0' when network play is not enabled.

We will now look at our basic DirectX 10 3D interactive environment's implementation (the core of our dynamically scalable interactive rendering engine as presented in Chapter 3). This basic implementation features mesh-loading, texture mapping, movable light sources, a GUI (one button for switching to full-screen mode) and stencil shadow volumes. The environment allows for full control of the camera – hence, the ability to move around freely. The core sections of the program are discussed here with the source code available on the accompanying CD.

2.4 Creating a Basic Interactive DirectX 10 3D Environment



The core implementation of the rendering engine created for this study (an example scene is shown in Figure 2.3) allows for basic input control in the form of user-movable light sources (the number only capped by hardware limitations), a six-directional moveable first-person camera and a movable mesh. It simply loads two meshes – one for the scene (the room) and one for the movable object (the drone). These meshes are provided as part of the DirectX software development kit and used here for the sake of convenience.



Please note: the drone was originally released as part of XNA in the .x file format and converted to the .sdkmesh format via the MeshConvert utility located in the DX10 SDK's "...\Utilities\Bin\x86" directory.

Figure 2.3 Our Direct3D 10 interactive environment.

Our implementation starts with the declaration of a structure to hold the coordinates and colour of a light source:

```
struct LightingProperties
{
    D3DXVECTOR3 Position; //a three-component vector - x, y, z
    D3DXVECTOR4 Colour; //a four-component vector - 3 colour values & alpha

    LightingProperties() {}

    LightingProperties(D3DXVECTOR3 Position_, D3DXVECTOR4 Colour_)
    {
        Position = Position_;
        Colour = Colour_;
    }
};
```

Next a structure is used to setup the spatial position and colour of a light source:

```
LightingProperties g_SetupLights = LightingProperties(D3DXVECTOR3(-4.0f, 1.0f, -4.0f),
                                                    D3DXVECTOR4(10.0f, 10.0f, 10.0f, 1.0f ));
```

It is also necessary to declare a structure to store the position, colour and world transformation matrix of a light source. This structure will be used to setup and translate a light source in 3D space using the previously declared `g_SetupLights` data:

```
struct LightData
{
    D3DXVECTOR3 m_LightPosition;
    D3DXVECTOR4 m_LightColour;
    D3DXMATRIX m_WorldTransformationMatrix;
};
```

This structure is used for the declaration of a `LightData` object that will store the data of each light source:

```
LightData g_LightObjectData;
```

The next step is to declare a number of global variables, starting with the main camera control variables (using the `DXUTcamera` helper class types – a class within Microsoft’s `DXUT` Framework, discussed in Appendix F, provided to simplify the management of view and projection transformations):



```
//first-person perspective model view camera
CFirstPersonCamera g_FPSModelViewCamera;

//camera for controlling the 3D mesh movement
CModelViewerCamera g_MeshControlCamera;

//camera for controlling light movement
CModelViewerCamera g_LightControlCamera;
```

Two matrices are also declared, the first to scale the object mesh (drone) and the second for scaling and translating the map mesh (room):

```
D3DXMATRIX g_MeshScalingMatrix;
D3DXMATRIX g_BackgroundWorldMeshMatrix;
```

Next a **D3DXVECTOR4** that will be used to set the scene's ambient lighting colour is declared:

```
D3DXVECTOR4 AmbientLighting(0.1f, 0.1f, 0.1f, 1.0f);
```

Input control is linked to three mouse buttons with the left mouse button controlling rotation of the viewer's camera, the middle mouse button controlling rotation of the light source and the right mouse button controlling the drone's rotation:

```
//true when the left mouse button is pressed
bool g_bLeftMBPressed = false;

//true when the right mouse button is pressed
bool g_bRightMBPressed = false;

//true when the middle mouse button is pressed
bool g_bMiddleMBPressed = false;
```

As our engine is DirectX based, a number of Direct3D 10 resources need to be declared (mesh objects, interfaces for managing vertex buffer and input layout objects, projection and view matrices, etc):

```
/* a mesh object used to read the background .sdkmesh files
   into memory */
CDXUTSDKMesh g_GameLevelMesh10;
```

```

/* a mesh object used to read the movable .sdkmesh file into
   memory */
CDXUTSDKMesh g_MeshObject;

//interface for implementing a rendering effect
ID3D10Effect* g_pID3D10Effect = NULL;

//interface managing a vertex buffer resource
ID3D10Buffer* g_pID3D10VertexBuffer = NULL;

//interface for a vertex input layout object
ID3D10InputLayout* g_pID3D10VertexLayout = NULL;

/* ID3D10EffectTechnique interfaces */
ID3D10EffectTechnique* g_pID3D10EffectRenderTextured = NULL;
ID3D10EffectTechnique* g_pID3D10EffectRenderLit = NULL;
ID3D10EffectTechnique* g_pID3D10EffectRenderAmbient = NULL;
ID3D10EffectTechnique* g_pID3D10EffectRenderShadow = NULL;

/* ID3D10EffectMatrixVariable interfaces for reading shader variables as matrix types*/
ID3D10EffectMatrixVariable* g_pd3d10ProjMatrixVar = NULL; //projection matrix
ID3D10EffectMatrixVariable* g_pd3d10ViewMatrixVar = NULL; //view matrix
ID3D10EffectMatrixVariable* g_pd3d10WorldMatrixVar = NULL; //world matrix

/* ID3D10EffectShaderResourceVariable interface for accessing
   shader-resource variables */
ID3D10EffectShaderResourceVariable* g_pd3d10DiffuseTexture = NULL;

/* ID3D10EffectVectorVariable interfaces for accessing shader
   variables as vector types */
ID3D10EffectVectorVariable* g_pd3d10LightPositionVectorVar = NULL;
ID3D10EffectVectorVariable* g_pd3d10LightColourVectorVar = NULL;
ID3D10EffectVectorVariable* g_pd3d10AmbientLightingVectorVar = NULL;
ID3D10EffectVectorVariable* g_pd3d10ShadowColourVectorVar = NULL;

/* ID3D10EffectScalarVariable interfaces for accessing scalar
   shader-resource variables */
ID3D10EffectScalarVariable* g_pd3d10ExtrudeShadowAmountScalarVar = NULL;
ID3D10EffectScalarVariable* g_pd3d10ExtrudeShadowBiasScalarVar = NULL;

```

With all these global variables set, the application's entry point, `wWinMain`, can now be defined. This function initialises the message processing loop elements and the idle time

required for the rendering of our scene. In the `wWinMain` function given below, several calls are made to various callback functions. These callback functions will be described later:

```
int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine,
                  int nCmdShow)
{
    /* set DXUT callbacks */
    //////////////////////////////////////

    /* set a callback function to change the device settings
       prior to device creation */
    DXUTSetCallbackDeviceChanging(SetD3D10DeviceSettings);

    //set the main message callback function
    DXUTSetCallbackMsgProc(MsgProc);

    //set the mouse event callback function
    DXUTSetCallbackMouse(MouseEventProcessing);

    //set the frame update callback function
    DXUTSetCallbackFrameMove(HandleSceneFrameUpdates);

    /* set the callback creating the Direct3D 10 resources not
       dependent on the back buffer */
    DXUTSetCallbackD3D10DeviceCreated(OnD3D10CreateDevice);

    /* set the callback creating the Direct3D 10 resources
       dependent on the back buffer */
    DXUTSetCallbackD3D10SwapChainResized(OnD3D10SwapChainResized);

    /* set the callback function releasing resources created
       by the OnD3D10ResizedSwapChain function */
    DXUTSetCallbackD3D10SwapChainReleasing(ReleaseSwapChain);

    /* set the callback function releasing resources created
       by the OnD3D10CreateDevice function */
    DXUTSetCallbackD3D10DeviceDestroyed(OnD3D10DestroyDevice);

    /* set the callback function rendering the scene on a per-frame basis */
    DXUTSetCallbackD3D10FrameRender(RenderFrame);

    Initialise(); //initialise the application
}
```



```
/* initialise DXUT: parses for command line arguments,  
   shows a message box on errors */  
DXUTInit(true, true, NULL);  
  
/* properties of the mouse cursor in full-screen mode (show  
   it & prevent it from exiting the screen boundaries) */  
DXUTSetCursorSettings(true, true);  
  
//create an application window with the specified caption  
DXUTCreateWindow(L"An Interactive Environment");  
  
/* create a Direct3D 10 device with an initial width and height */  
DXUTCreateDevice(true, 1024, 768);  
  
//enter the main DXUT execution loop  
DXUTMainLoop();  
  
return DXUTGetExitCode();  
}
```

The `DXUTSetCallbackDeviceChanging` DXUT function sets a callback function responsible for changing the device settings prior to device creation. The `SetD3D10DeviceSettings` callback function is passed as parameter and used for this purpose (specifying how to create the D3D10 device):

```
bool CALLBACK SetD3D10DeviceSettings(DXUTDeviceSettings* pDeviceSettings,  
                                     void* pUserContext)  
{  
    /* the DXGI_FORMAT_D24_UNORM_S8_UINT format supports stencilling */  
    pDeviceSettings->d3d10.AutoDepthStencilFormat = DXGI_FORMAT_D24_UNORM_S8_UINT;  
  
    return true;  
}
```

The next callback function, `MsgProc` (passed as parameter to the `DXUTSetCallbackMsgProc` DXUT initialisation function) handles all application messages. This callback function is called whenever an event occurs and it is declared as follows:

```
LRESULT CALLBACK MsgProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam,  
                        bool* pbNoFurtherProcessing, void* pUserContext)
```



```

{
    /* first let the dialogues handle all generated messages before passing on the
       remaining messages - see full program source code for details */

    /* all remaining messages (user input) should be passed to the camera */
    g_FPSTModelViewCamera.HandleMessages(hWnd, uMsg, wParam, lParam);
    g_MeshControlCamera.HandleMessages(hWnd, uMsg, wParam, lParam);
    g_LightControlCamera.HandleMessages(hWnd, uMsg, wParam, lParam);

    return 0;
}

```

The mouse event callback function, `MouseEventProcessing`, processing all mouse input, is subsequently set in the `wWinMain` function via the `DXUTSetCallbackMouse` DXUT function:

```

void CALLBACK MouseEventProcessing(bool bLeftButtonDown,
                                  bool bRightButtonDown,
                                  bool bMiddleButtonDown,
                                  bool bSideButton1Down,
                                  bool bSideButton2Down,
                                  int nMouseWheelDelta,
                                  int xPos, int yPos,
                                  void* pUserContext)
{
    /* flags indicating the mouse buttons pressed */
    bool bOldLeftButtonDown = g_bLeftMBPressed;
    bool bOldRightButtonDown = g_bRightMBPressed;
    bool bOldMiddleButtonDown = g_bMiddleMBPressed;

    g_bLeftMBPressed = bLeftButtonDown; //is the left mouse button down?
    g_bMiddleMBPressed = bMiddleButtonDown; //is the middle mouse button down?
    g_bRightMBPressed = bRightButtonDown; //is the right mouse button down?

    //move the mesh if the right mouse button is down
    if(bOldRightButtonDown && !g_bRightMBPressed)
    {
        g_MeshControlCamera.SetEnablePositionMovement(false);
    }
    else
        if(!bOldRightButtonDown && g_bRightMBPressed)
        {
            g_MeshControlCamera.SetEnablePositionMovement(true);
        }
}

```

```

        g_FPSModelViewCamera.SetEnablePositionMovement(false);
    }

    //rotate the player camera if the left mouse button is down
    if(bOldLeftButtonDown && !g_bLeftMBPressed)
        g_FPSModelViewCamera.SetEnablePositionMovement(false);
    else
        if(!bOldLeftButtonDown && g_bLeftMBPressed)
            g_FPSModelViewCamera.SetEnablePositionMovement(true);

    //move the light source if the middle mouse button is down
    if(bOldMiddleButtonDown && !g_bMiddleMBPressed)
    {
        g_LightControlCamera.SetEnablePositionMovement(false);
    }
    else
        if(!bOldMiddleButtonDown && g_bMiddleMBPressed)
        {
            g_LightControlCamera.SetEnablePositionMovement(true);
            g_FPSModelViewCamera.SetEnablePositionMovement(false);
        }

    /* move the player camera if none of the mouse buttons are held down */
    if(!g_bRightMBPressed && !g_bMiddleMBPressed && !g_bLeftMBPressed)
        g_FPSModelViewCamera.SetEnablePositionMovement(true);
}

```

The frame update callback function, `HandleSceneFrameUpdates`, processing each scene update, is set by the `DXUTSetCallbackFrameMove` DXUT function and defined as follows:

```

void CALLBACK HandleSceneFrameUpdates(double time, float timePassed, void* context)
{
    /* update the view matrix based on user input and elapsed time */
    g_FPSModelViewCamera.FrameMove(timePassed);
    g_MeshControlCamera.FrameMove(timePassed);
    g_LightControlCamera.FrameMove(timePassed);
}

```

The callback function creating the Direct3D 10 resources not dependent on the back buffer, `OnD3D10CreateDevice`, is set via the `DXUTSetCallbackD3D10DeviceCreated` DXUT function and defined as follows:

```

HRESULT CALLBACK OnD3D10CreateDevice(ID3D10Device* pd3dDevice,
                                     const DXGI_SURFACE_DESC *pBackBufferSurfaceDesc,
                                     void* pUserContext)
{
    //the effect file
    WCHAR effectName[MAX_PATH];

    //read and compile the effect
    DXUTFindDXSDKMediaFileCch(effectName, MAX_PATH, L"MainFX10.fx");

    //create an effect from the file
    D3DX10CreateEffectFromFile(effectName, NULL, NULL, "fx_4_0",
                               D3D10_SHADER_ENABLE_STRICTNESS,
                               0, pd3dDevice, NULL, NULL,
                               &g_pID3D10Effect, NULL, NULL);

    /* get the technique handles by name from the MainFX10.fx file */
    g_pID3D10EffectRenderTextured = g_pID3D10Effect->
        GetTechniqueByName("RenderTextured");
    g_pID3D10EffectRenderLit = g_pID3D10Effect->
        GetTechniqueByName("RenderLitEnvironment");
    g_pID3D10EffectRenderAmbient = g_pID3D10Effect->
        GetTechniqueByName("RenderWithAmbientLighting");
    g_pID3D10EffectRenderShadow = g_pID3D10Effect->
        GetTechniqueByName("RenderSceneWithShadow");

    /* create the input-assembler stage's single element description */
    const D3D10_INPUT_ELEMENT_DESC vertex_input_layout[] =
    {
        {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0},
        {"TEXTURE", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24, D3D10_INPUT_PER_VERTEX_DATA, 0},
        {"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D10_INPUT_PER_VERTEX_DATA, 0},
    };

    //structure to describe each effect pass
    D3D10_PASS_DESC EffectPassDescription;

    //get the effect pass to render the scene lit
    g_pID3D10EffectRenderLit->GetPassByIndex(0)
        ->GetDesc(&EffectPassDescription);

    //create an input-layout object
    pd3dDevice->CreateInputLayout(vertex_input_layout, 3,

```



```
        EffectPassDescription.pIAInputSignature,  
        EffectPassDescription.IAInputSignatureSize,  
        &g_pID3D10VertexLayout);  
  
    /* load the mesh representing the environment/game map as  
       well as the character mesh */  
    g_GameLevelMesh10.Create(pd3dDevice, L"\\blackholeroom.sdkmesh", false, true);  
    g_MeshObject.Create(pd3dDevice, L"\\EvilDrone.sdkmesh", false, true);  
  
    //get the effect variables by name (from MainFX10.fx)  
    g_pd3d10ProjMatrixVar = g_pID3D10Effect->  
        GetVariableByName("ProjectionMatrix")->AsMatrix();  
    g_pd3d10ViewMatrixVar = g_pID3D10Effect->  
        GetVariableByName("ViewMatrix")->AsMatrix();  
    g_pd3d10WorldMatrixVar = g_pID3D10Effect->  
        GetVariableByName("WorldMatrix")->AsMatrix();  
    g_pd3d10DiffuseTexture = g_pID3D10Effect->  
        GetVariableByName("DiffuseTexture")->AsShaderResource();  
    g_pd3d10LightPositionVectorVar = g_pID3D10Effect->  
        GetVariableByName("LightPosition")->AsVector();  
    g_pd3d10LightColourVectorVar = g_pID3D10Effect->  
        GetVariableByName("LightColour")->AsVector();  
    g_pd3d10AmbientLightingVectorVar = g_pID3D10Effect->  
        GetVariableByName("AmbientLighting")->AsVector();  
    g_pd3d10ShadowColourVectorVar = g_pID3D10Effect->  
        GetVariableByName("ShadowColour")->AsVector();  
    g_pd3d10ExtrudeShadowAmountScalarVar = g_pID3D10Effect->  
        GetVariableByName("ShadowExtrusionAmount")->AsScalar();  
    g_pd3d10ExtrudeShadowBiasScalarVar = g_pID3D10Effect->  
        GetVariableByName("ShadowExtrusionBias")->AsScalar();  
  
    /* set the camera at the centre of projection (eye) pointed  
       towards the "at" location */  
    D3DXVECTOR3 eye(0.0f, 3.0f, -8.0f);  
    D3DXVECTOR3 at(0.0f, 3.1f, 0.0f);  
    g_FPSModelViewCamera.SetViewParams(&eye, &at);  
    g_LightControlCamera.SetViewParams(&eye, &at);  
    g_MeshControlCamera.SetViewParams(&eye, &at);  
  
    return S_OK;  
}
```

The callback function creating the Direct3D 10 resources dependent on the back buffer, `OnD3D10SwapChainResized`, is set using the `DXUTSetCallbackD3D10SwapChainResized` DXUT function. This function, called for each swap chain resize is given here (the swap chain, as discussed in Appendix F, is used to display the contents of either the front or back buffer):

```
HRESULT CALLBACK OnD3D10SwapChainResized(ID3D10Device* pd3dDevice,
                                         IDXGISwapChain *pSwapChain,
                                         DXGI_SURFACE_DESC* pBackBufferSurfaceDesc,
                                         void* pUserContext)
{
    //calculate aspect ratio
    float WidthHeightRatio = pBackBufferSurfaceDesc->
        Width/(FLOAT)pBackBufferSurfaceDesc->Height;

    /* called the moment the Direct3D 10 swap chain is about to
       be resized or created */
    g_DXUTDialogResourceManager.OnD3D10ResizedSwapChain(
        pd3dDevice, pBackBufferSurfaceDesc);

    //set the camera's projection parameters
    g_FPSModelViewCamera.SetProjParams(D3DX_PI/4, WidthHeightRatio, 0.1f, 500.0f);
    g_MeshControlCamera.SetWindow(pBackBufferSurfaceDesc
        ->Width, pBackBufferSurfaceDesc->Height);
    g_LightControlCamera.SetWindow(pBackBufferSurfaceDesc
        ->Width, pBackBufferSurfaceDesc->Height);

    return S_OK;
}
```

The resources created in these `OnD3D10ResizedSwapChain` and `OnD3D10CreateDevice` functions are subsequently released by the `ReleaseSwapChain` and `OnD3D10DestroyDevice` callback functions (set in `wWinMain` using the `DXUTSetCallbackD3D10SwapChainReleasing` and `DXUTSetCallbackD3D10DeviceDestroyed` DXUT functions, respectively). See the full source code available on the included CD for the related definitions.

We set the callback function rendering the scene on a per-frame basis by means of the `DXUTSetCallbackD3D10FrameRender` DXUT initialisation function. This callback, `RenderFrame`, renders the complete frame (all the meshes, shadows, lights, etc). The `RenderFrame` function is given here:

```

void CALLBACK RenderFrame(ID3D10Device* pd3dDevice, double fTime, float fElapsedTime,
                          void* pUserContext)
{
    //set the clear colour to black
    float RenderTargetClearColour[4] = {0.0, 0.0, 0.0, 0.0};

    //clear the render target
    ID3D10RenderTargetView* pRenderTargetView = DXUTGetD3D10RenderTargetView();
    pd3dDevice->ClearRenderTargetView(pRenderTargetView, RenderTargetClearColour);

    //clear the stencil buffer
    ID3D10DepthStencilView* pDepthStencilView = DXUTGetD3D10DepthStencilView();
    pd3dDevice->ClearDepthStencilView(pDepthStencilView, D3D10_CLEAR_DEPTH, 1.0f, 0);

    //bind the input-layout object to the input-assembler stage
    pd3dDevice->IASetInputLayout(g_pID3D10VertexLayout);

    //draw the scene with ambient lighting
    g_pd3d10AmbientLightingVectorVar->SetFloatVector((float*)&AmbientLighting);
    RenderScene(pd3dDevice, g_pID3D10EffectRenderAmbient, false);

    /* set the amount and bias to extrude the shadow volume from the silhouette edge*/
    g_pd3d10ExtrudeShadowAmountScalarVar->SetFloat(120.0f - 0.1f);
    g_pd3d10ExtrudeShadowBiasScalarVar->SetFloat(0.1f);

    /* setup the light */
    D3DXVECTOR4 LightVector(g_LightObjectData.m_LightPosition.x,
                           g_LightObjectData.m_LightPosition.y,
                           g_LightObjectData.m_LightPosition.z,
                           1.0f);

    D3DXVec4Transform(&LightVector, &LightVector,
                      g_LightControlCamera.GetWorldMatrix());
    g_pd3d10LightPositionVectorVar->SetFloatVector((float*)&LightVector);
    g_pd3d10LightColourVectorVar->SetFloatVector(
        ((float*)g_LightObjectData.m_LightColour);

    /*for the light source, render the resulting shadow*/
    //////////////////////////////////////

    //clear the stencil buffer
    pd3dDevice->ClearDepthStencilView(pDepthStencilView, D3D10_CLEAR_STENCIL, 1.0, 0);

```



```
//prepare to render the shadow volume
ID3D10EffectTechnique* pEffectTechnique = g_pID3D10EffectRenderShadow;

//render the actual shadow
RenderScene(pd3dDevice, pEffectTechnique, true);

//render the scene with normal lighting
RenderScene(pd3dDevice, g_pID3D10EffectRenderLit, false);

/* code to render the GUI - the "Full-Screen Mode" button
   - see full program source code for details */

DXUT_EndPerfEvent();
}
```

The `RenderFrame` function calls the `RenderScene` function when drawing the scene with ambient lighting (without shadows), when rendering the drone's shadow and when rendering the final lit and shadowed scene. The `RenderScene` function renders the map/level mesh, the drone character and the shadow. It is also responsible for calculating the view matrices:

```
void RenderScene(ID3D10Device* pd3dDevice, ID3D10EffectTechnique* pEffectTechnique,
                bool renderShadowVol)
{
    //setup the view matrices
    D3DXMATRIX ProjectionMatrix;
    D3DXMATRIX ViewMatrix;
    D3DXMATRIX ViewProjectionMatrix;
    D3DXMATRIX WorldMatrix;
    D3DXMATRIX WorldViewProjectionMatrix;

    //calculate the projection matrix
    ProjectionMatrix = *g_FPSModelViewCamera.GetProjMatrix();
    //calculate the view matrix
    ViewMatrix = *g_FPSModelViewCamera.GetViewMatrix();

    //calculate and set the view project matrix
    ViewProjectionMatrix = ViewMatrix * ProjectionMatrix;
    g_pd3d10ViewMatrixVar->SetMatrix((float*)&ViewProjectionMatrix);

    /* render the mesh representing the map/level */
    if(!renderShadowVol)
```



```

{
    //calculate and set the world view projection matrix
    WorldViewProjectionMatrix = g_BackgroundWorldMeshMatrix *
        ViewMatrix *
        ProjectionMatrix;
    g_pd3d10ProjMatrixVar->SetMatrix((float*)&WorldViewProjectionMatrix);
    g_pd3d10WorldMatrixVar->SetMatrix((float*)&g_BackgroundWorldMeshMatrix);

    //render the map mesh
    g_GameLevelMesh10.Render(pd3dDevice, pEffectTechnique, g_pd3d10DiffuseTexture);
}

/* render the mesh representing the object/character */
////////////////////////////////////

//calculate the world matrix
WorldMatrix = g_MeshScalingMatrix * g_MeshControlCamera.GetWorldMatrix();
//calculate the world view projection matrix
WorldViewProjectionMatrix = WorldMatrix * ViewMatrix * ProjectionMatrix;
//set the world and world view project matrices
g_pd3d10ProjMatrixVar->SetMatrix((float*)&WorldViewProjectionMatrix);
g_pd3d10WorldMatrixVar->SetMatrix((float*)&WorldMatrix);

//render the character mesh and the shadow
if(renderShadowVol)
    g_MeshObject.RenderAdjacent(pd3dDevice,pEffectTechnique,g_pd3d10DiffuseTexture);
else
    g_MeshObject.Render(pd3dDevice, pEffectTechnique, g_pd3d10DiffuseTexture);
}

```

All that remains now is to initialise the application. This is done in `wWinMain` via a call to our own `Initialise` function:

```

void Initialise()
{

    /* init the application HUD (the "Full-Screen Mode" button) - see full program
    source code for details */

    //init the light
    g_LightObjectData.m_LightPosition = g_SetupLights.Position;
    g_LightObjectData.m_LightColour = g_SetupLights.Colour;
}

```

```

//initialise the cameras
g_FPSModelViewCamera.SetRotateButtons(true, false, false);
g_MeshControlCamera.SetButtonMasks(MOUSE_RIGHT_BUTTON,0,0);
g_LightControlCamera.SetButtonMasks(MOUSE_MIDDLE_BUTTON,0,0);
/* scale and translate the environment's map mesh */
////////////////////////////////////

//the translation matrix
D3DXMATRIX mapTranslationMatrix;

//create the translation matrix
D3DXMatrixTranslation(&g_BackgroundWorldMeshMatrix,0.0f,1.0f, 0.0f);
D3DXMatrixTranslation(&mapTranslationMatrix, 1.0f, 1.0f, 0.0f);

//create an identity matrix
D3DXMatrixIdentity(&g_MeshScalingMatrix);
}

```

The presented game engine's main application code has now been discussed. The next chapter focuses on this engine's extension through the addition of several subsystems, specifically: HLSL shaders, local illumination, reflection and refraction, HDR lighting, additional shadow rendering algorithms, physics simulation, particle effects and post-processing special effects.

2.5 Summary

The chapter started by looking at game engine architecture in general, highlighting the importance of software componentry, and the difference between game-engine code and game-specific code. Following this it focussed on a number of game engine architectures, specifically ad-hoc, modular and the directed acyclic graphs architecture (DAG).

Next it considered the first step invoked whenever a game is executed, namely initialisation. Initialisation was described as the stage responsible for resource and device acquisition, memory allocation, setup of the game's GUI, loading of art assets, etc. Following front-end initialisation, it discussed the exit state and the game loop for the uninterrupted execution of a game.

Building on this, the chapter dealt with the implementation of a basic DirectX 10 3D interactive environment featuring mesh-loading, texture mapping, movable light sources, a GUI and stencil shadow volumes; the core platform upon which more advance engine features are to be layered (presented in Chapter 3).

Extending the Basic Interactive 3D Environment

Chapter 3 builds on Chapter 2's basic DirectX 10 3D interactive environment featuring mesh-loading, texture mapping, movable light sources, a GUI and stencil shadow volumes. The presented 3D environment is then extended through the addition of several subsystems, specifically: HLSL shaders, local illumination, reflection and refraction, HDR lighting, additional shadow rendering algorithms, physics simulation, particle effects and post-processing special effects. Part II of this thesis categorises each of these subsystems based on the level-of-detail/rendering quality and the associated computational impact.

Outline:

- Extending the presented interactive DirectX 10 3D environment
- Shaders
- Local Illumination
- Reflection and Refraction
- High Dynamic Range Lighting
- Shadows
- Physics
- Particle Effects
- Post-Processing

3.1 Extending the Basic Interactive DirectX 10 3D Environment

The modular rendering engine developed for this study and serving as a scalable interactive testing environment is an adequate platform for the purposes of this thesis, in which the objective is to experiment with the impact of various algorithms when rendering computationally intensive 3D environments solution for the rendering of computationally intensive 3D environments. As a standalone game engine, it is amenable to being used as a game engine for first- and third-person shooter games, role playing games and 3D immersive environments. The engine makes extensive use of DirectX 10.0 and Shader Model 4.0 (a proprietary shading language developed for use with the Direct3D API) for effects such as HDR and dynamic ambient lighting, volumetric clouds, motion blur, soft shadows, specular reflections, reflective and refractive water, motion blur, etc. The engine also features support for high polygon models, realistic physics and particle effects. Figure 3.1 shows the further extended interactive environment/rendering engine.

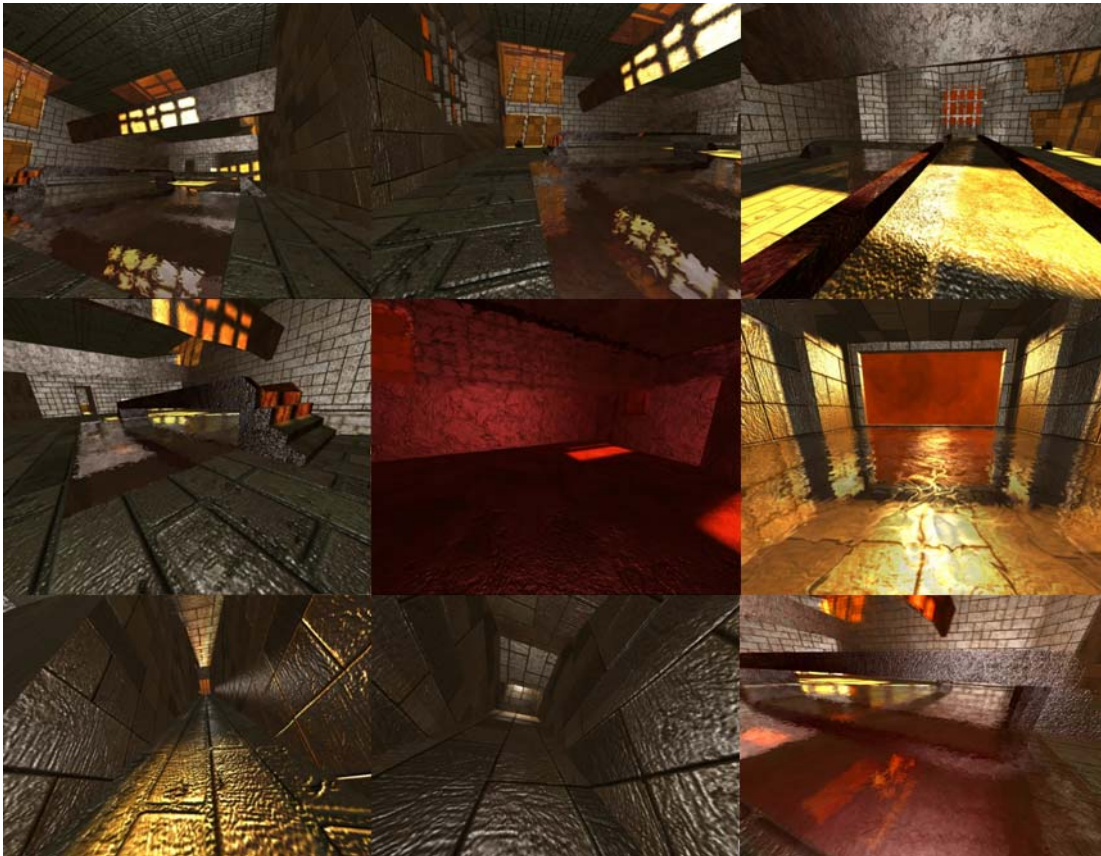


Figure 3.1 Various screenshots of the extended interactive testing environment.

The testing environment's technology stack utilises SIMD and multi-core processor technologies, as well as HLSL Shader Model 4.0 and the latest DirectX GPU features. The quality of rendering elements is dynamically scalable based on GPU (and to a limited degree, CPU) usage and under/over-utilisation. For example, shader quality

relies on the GPU, ranging from low (simplified shaders, light maps and directional lights), medium (simplified HDR, normal maps and specular highlights), high (soft shadows, detail lights, ambient occlusion and soft particles) to very high (true HDR, translucent shadows, parallax mapping and volumetric materials). The quality of particle effects, in turn, relies on the CPU and can range from low with a 75% reduction in quality to very high with no reduction in quality.

The purpose of the testing environment is to serve as a proof of concept platform to test the feasibility of two different but related performance enhancing strategies. The first may be termed quality scaling, by which is meant controlling the quality of a rendered scene by selecting algorithms that provide an appropriate level of realism for the given context. The second performance enhancing strategy is the provision of a mechanism that will seamlessly and in real-time ensure quality scaling by dynamically activating the appropriate set of rendering algorithms as a scene changes. Using this data gathered during the performance vs. quality analysis of this platform, we are able to control the real-time selection of rendering algorithms based on environmental conditions. This system ensures the following: the quality of the scene being rendered is always maximised with the GPU and CPU unified as single rendering unit for the maximised processing of reflections, particle effects and physics simulations.

The sections below detail the presented interactive testing environment's core rendering and/or computational elements – shadows, shaders, local illumination, reflection and refraction, physics, particle effects and post-processing (including implementation details). However, Appendix A can be consulted should background information be needed on the concept of programmable pipelines (and the graphics pipeline architecture, in general) as well as on how these processing pipelines allow for the direct manipulation of the movement, composition, form and appearance of objects – aspects integral to any modern 3D rendering engine design and implementation. Appendix A topics include:

- Vertex Processing
- Clipping and Culling
- Rasterization and Fragment Processing
- Programmable Pipelines
 - The Direct3D 10 Processing Pipeline
 - The Input-Assembler Stage
 - The Geometry-Shader Stage
 - The Vertex-Shader Stage
 - Stream Output Stage
 - The Pixel-Shader Stage
 - The Output-Merger Stage

Microsoft's High Level Shader Language is also dealt with throughout many of the interactive testing environment implementations that follow, such as those dealing with HDR lighting, stencil shadow volumes, bump mapping, cube mapping and motion blur (illustrating geometric shaders), adding specular highlights to objects, etc. Appendix B presents shaders (an integral part of our engine's ability to render hyper-realistic 3D environments and briefly touched on in Section 3.2) in detail – specific topics dealt with include:

- Shaders
- The Hardware Graphics Pipeline
- The Programmable Graphics Pipeline Revisited
- The High Level Shader Language (HLSL)
 - The HLSL Compiler
 - Initialising the High Level Shader Language
 - Creating HLSL Shaders
 - Common HLSL Data Types
 - Utilising a Created HLSL Effect

As previously mentioned, the presented dynamically scalable interactive rendering engine (serving as proof-of-concept and benchmarking system) features a number of advanced rendering components, specifically: shaders, lighting, reflection and refraction, shadows, physics, particles and post-processing effects. Each of these can, in turn, be categorised based on the level-of-detail/rendering quality and the associated computational impact (discussed in Part II of this thesis). The implementation details of these rendering features used to extend the basic interactive environment presented in Section 2.4 are now discussed.

Please note that all rendering techniques are implemented in C++ using Direct3D 10.0 and Microsoft's High Level Shading Language 4.0. Subsequent sections illuminate 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.

3.2 Shaders

A shader is a grouping of instructions processed by the graphics accelerator to perform some form of special effect or rendering. Appendix A presents the concept of programmable pipelines (in particular focusing on the Direct3D 10 and OpenGL processing pipelines). An application program allowing direct interaction with these programming pipelines is called a *shader*. Shader programs, written in a shading language such as NVIDIA's Cg or Microsoft's High Level Shader Language, control the

movement, composition, form and appearance of objects through direct manipulation of the graphics processing unit's programmable pipelines (Fernando and Kilgard, 2003).

The instructions listed in a shader program are executed at a specific point in the rendering pipeline – thus leading to user-defined manipulation of vertex or pixel data, for example. More specifically, three types of shader programs can be written, namely, vertex shaders, pixel shaders and geometry shaders.

Vertex shaders, operating on vertex data, are executed as part of the graphics pipeline's geometric stage and are used to alter the geometric parameters (shape) of an object. A vertex shader program is fundamental for certain special effects such as grass blowing in the wind where the real time manipulation, transformation and displacement of per-vertex material attributes are necessary. The vertices produced by this shader are forwarded as input to a geometry shader.

Geometry shaders are executed just prior to the rasterizer and stream output pipeline stages. These shaders group numerous vertices into a geometric object that can be modified by a pixel shader program. Geometry shaders are extremely important in the detection of silhouetted-edges and shadow volume extrusion. These shaders, performing per-primitive (low-level geometric objects such as points, lines, etc.) computations, are also vital in the generation of new primitives. The primitives generated by the geometry shader stage are rasterized into fragments during the pipeline's rasterizer stage. These fragments are then sent to the pixel shader as input.

Pixel shaders, also known as *fragment shaders* and performing per-pixel processing, operate on the discrete pixels of a primitive, applying some effect to a primitive (such as bump mapping, shadowing, fog, etc) during the pixel shader stage. Per-pixel lighting and shadowing has greatly contributed to the realism of modern computer games. Examples of effects made possible through this form of per-pixel processing include texture blending, environmental mapping, normal mapping, real-time shadows (stencil shadow volumes) and reflections.

These three types of shaders are unified by the Direc3D 10 architecture – known as Shader Model 4.0. *Unified shaders* provide the application programmer with a uniform instruction set that is independent of whether a pixel, geometry or vertex shader is being implemented. This unified architecture is made possible through Windows Vista's and Windows 7's Windows Display Driver Model and the coupled DirectX 10 API. Previous architectures required different instruction sets for both pixel and vertex shaders due to specific hardware architectural requirements. By unifying the independent shader instruction sets, GPU programming has become much more flexible. This unified model also allows workload sharing amongst the various pipeline processors. For example, when the GPU is mainly performing basic geometry rendering with little or no per-pixel processing being done, then the pixel shader can be assigned vertex processing. The

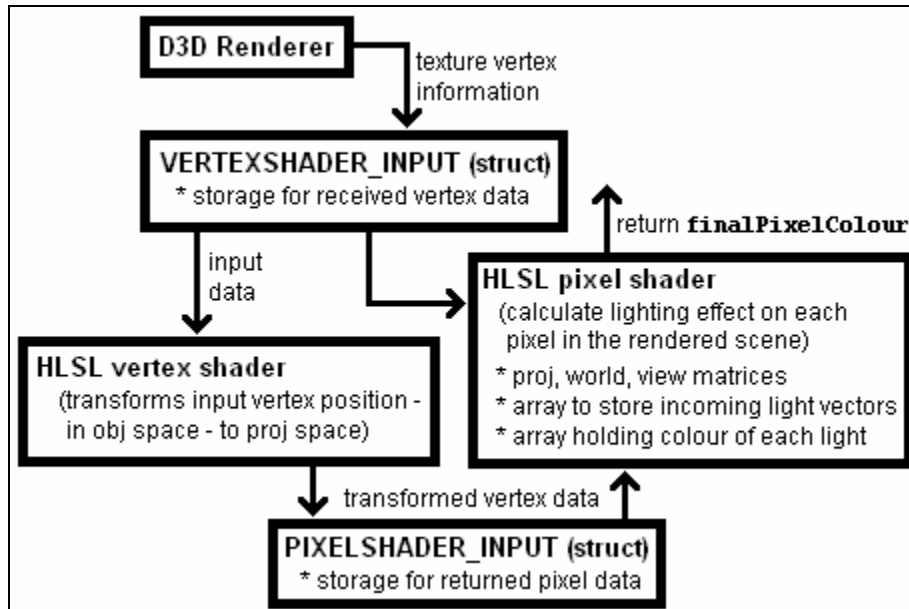
first GPU offering support for this unified shader model was NVIDIA's GeForce 8 series – specifically the GeForce 8800 GTX and GTS.

The term used to describe this unified shader architecture, Shader Model 4.0, encapsulates the features offered by the specific shader version in question. For example, Shader Model 3.0 (as supported by DirectX 9.0c) limits the number of executing instructions to 65536 while DirectX 10's Shader Model 4.0 allows for an unlimited number of executing instructions. Shader Model 2.0 (the original DirectX 9.0 shader specification) limits the number of executing instructions to 32 texture instructions and 64 arithmetic instructions. The version number of instructions is specified in terms of the shader's version number (`ps_mainVersion_subVersion` for pixel shaders and `vs_mainVersion_subVersion` for vertex shaders). For example, a vertex shader based on Shader Model 3.0 (DirectX 9.0c) will be declared as `vs_3_0`, a DirectX 9.0b Shader Model 2.0 pixel shader as `ps_2_b`, and a Shader Model 4.0 pixel shader declared as `ps_4_0`. NVIDIA's GeForce FX series of GPUs provide an optimised model for Shader Model 2.0 and we can thus define a vertex shader based on this model as `vs_2_a`.

Advanced shader technology, as further detailed in Appendix B, is core to the creation of realistic 3D environments, as the case with the presented dynamically scalable rendering engine. Shader technology is used for everything from controlling the geometric level-of-detail on model and world-elements to the anti-aliasing of alpha-tested primitives and the use of distance-coded alpha masking for infinite resolution texture masking when dealing with alpha-tested primitives and resolution-independent user interface elements. In the rest of this chapter, the deployment of shader technology for various rendering effects is shown in detail. The presentation is as follows:

- Local Illumination (Section 3.3)
- Reflection and Refraction (Section 3.4)
- High Dynamic Range Lighting (Section 3.5)
- Shadows (Section 3.6)
- Particle Effects (Section 3.7)
- Post-Processing (Section 3.8)

3.3 Local Illumination



The presented interactive testing environment, in its most basic form, allows for the use of local illumination which, unlike global illumination, only considers the interaction between a light source and object. For example, when lighting a series of cubes, each cube is lit independently from the others. Thus, even though one cube might be blocking light from another, the effect of this is never considered by the local illumination model (shadowing is thus only added at a later stage). This model is shown in Figure 3.2.

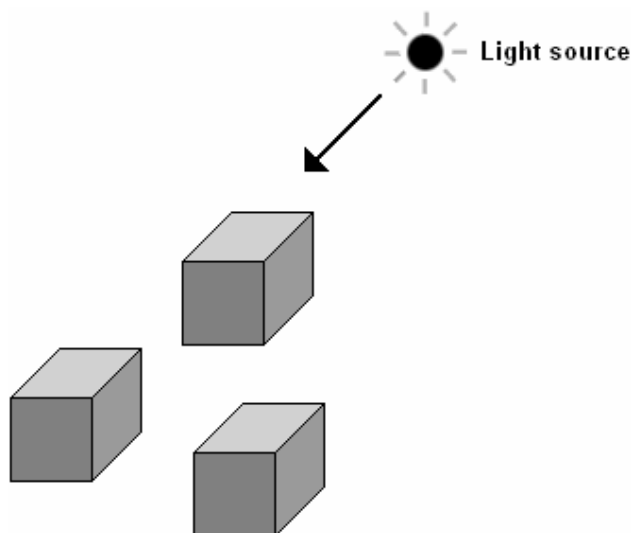


Figure 3.2 The local illumination model.

Global illumination, on the other hand, accounts for this “blocked-out light” via the implementation of a ray tracing algorithm, for example (Rubin and Whitted, 1980). Ray

tracing follows the light (via vectors) from the source to object surfaces, rendering objects and effects based on the subsequent object-light interaction (Arvo and Kirk, 1987). Global illumination is not supported in the rendering engine that has been implemented, as it falls outside the scope of interactive graphics, rather belonging to the field of photo realistic rendering (Arvo, 1991). Its overall effect can, however, be simulated through the use of a number of shadowing and reflection algorithms as discussed in subsequent sections. Figure 3.3 shows global illumination where one object blocks light from reaching other objects.

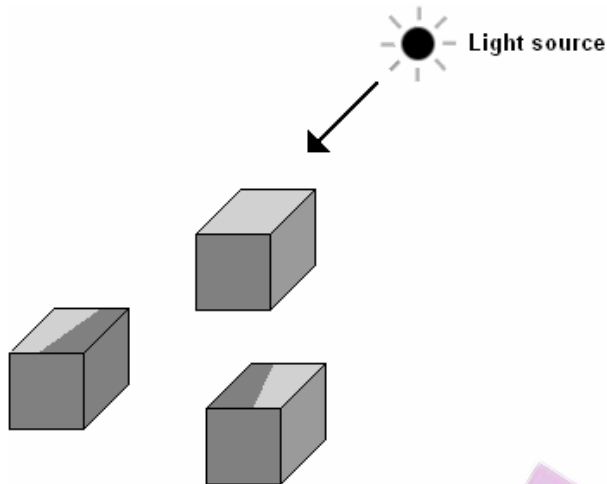


Figure 3.3 The global illumination model.

We implement local illumination using the diffuse reflection model, resulting in a uniformly lit scene. The amount of reflection is calculated using Lambert's law – hence by considering the cosine of the angle between the vector directed at the light source and the surface normal (Figure 3.4). The angle, θ , can be determined by calculating the dot product of these two vectors (Cook and Torrance, 1982).

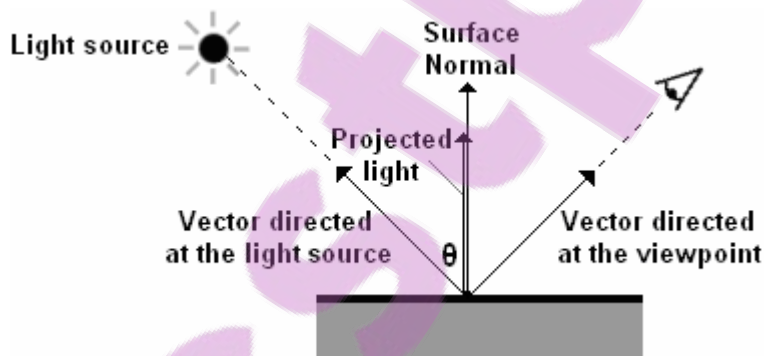


Figure 3.4 The projected light calculated by considering the cosine of the angle between the vector directed at the light source and the surface normal.

The rendered scene, comprised of several cubes, two parallel light sources and using Lambertian light, will thus have a consistent lighting intensity regardless of the distance between the reflecting surface and light source (as shown in Figure 3.5).

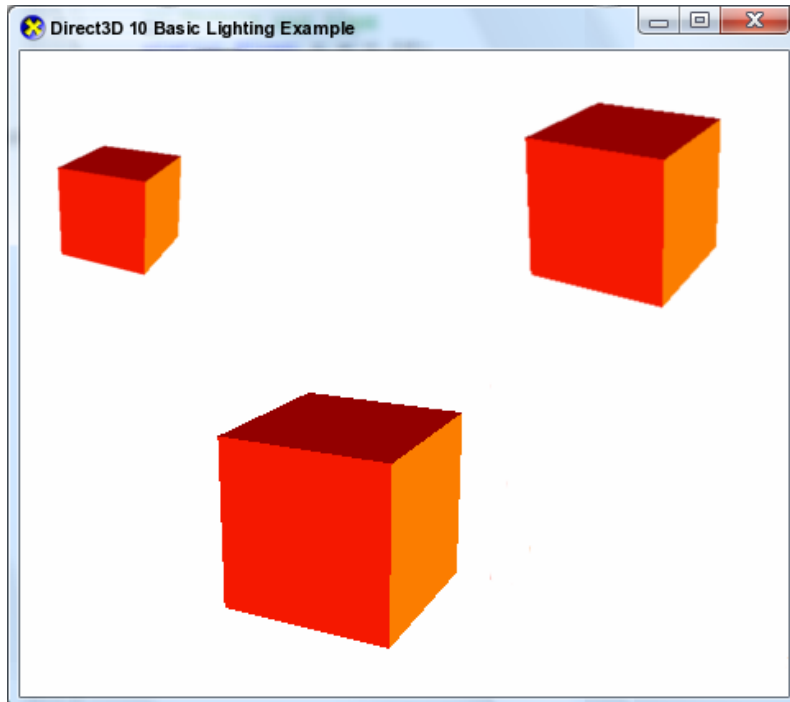


Figure 3.5 The rendered scene; comprised of three cubes, two parallel light sources and using Lambertian light.

The presented rendering engine implements an HLSL pixel shader to calculate the lighting effect on each pixel in the rendered scene. The shader's effect file starts with a declaration of the projection, world and view matrices followed by a floating point array storing the incoming light vector of each light source and another floating point array holding the colour of each light:

```
matrix ProjectionMatrix;  
  
matrix WorldMatrix;  
matrix ViewMatrix;  
  
float4 LightDirection[2];  
float4 LightColour[2];
```

These variables, declared using the HLSL data types, are set by the Direct3D application. We must thus declare variables in our application that will be used to update the shader variables:

```
D3DXMATRIX g_ProjectionMatrix;
D3DXMATRIX g_WorldMatrix;
D3DXMATRIX g_ViewMatrix;

D3DXVECTOR4 IncomingLightVector[2];
D3DXVECTOR4 IncomingLightColour[2];
```

The Direct3D application initialises these variables, subsequently binding them within the technique. The lighting position and colour arrays are set in the following manner:

```
/* initialise the direction of each parallel light source */
D3DXVECTOR4 IncomingLightVector[2] =
{
    //the spatial position of the first light source
    D3DXVECTOR4(1.0f, 0.5f, 0.5f, 1.0f),
    //the spatial position of the second light source
    D3DXVECTOR4(0.0f, 0.0f, 1.0f, 1.0f)
};

/* specify the colour of each parallel light source */
D3DXVECTOR4 IncomingLightColour[2] =
{
    //bright red
    D3DXVECTOR4(1.0f, 0.0f, 0.0f, 1.0f),
    //deep orange
    D3DXVECTOR4(1.0f, 0.5f, 0.0f, 1.0f)
};
```

The `g_WorldMatrix` variable is initialised to an identity matrix using the `D3DXMatrixIdentity` D3DX math function. The `g_ViewMatrix` variable is initialised via the `D3DXMatrixLookAtLH` D3DX function. The `g_ProjectionMatrix` variable is initialised using the `D3DXMatrixPerspectiveFovLH` D3DX function:

```
/* initialise the world matrix */
D3DXMatrixIdentity(&g_WorldMatrix);

/* initialise the view matrix */
D3DXVECTOR3 EyeCoord( 0.0f, 1.0f, -10.0f);
D3DXVECTOR3 LookAt(0.0f, 1.0f, 0.0f);
D3DXVECTOR3 UpDir(0.0f, 1.0f, 0.0f);
D3DXMatrixLookAtLH(&g_ViewMatrix, &EyeCoord, &LookAt, &UpDir);

/* set the left-handed perspective projection */
```

```
D3DXMatrixPerspectiveFovLH(&g_ProjectionMatrix, (float)D3DX_PI*0.25f,
                           rectangle_width/rectangle_height, 0.1f, 100.0f);
```

Before the `ID3D10EffectVariable` update methods can be used to set the HLSL variable values, we first have to obtain the effect variables via `ID3D10Effect` retrieval functions for each of the above defined shader variables:

```
/* obtain the ProjectionMatrix shader variable */
ID3D10EffectMatrixVariable* g_pd3d10ProjMatrixVar = NULL;
g_pd3d10ProjMatrixVar = g_pd3d10Effect
    ->GetVariableByName("ProjectionMatrix")->AsMatrix();

/* obtain the WorldMatrix shader variable */
ID3D10EffectMatrixVariable* g_pd3d10WorldMatrixVar = NULL;
g_pd3d10WorldMatrixVar = g_pd3d10Effect
    ->GetVariableByName("WorldMatrix")->AsMatrix();

/* obtain the ViewMatrix shader variable */
ID3D10EffectMatrixVariable* g_pd3d10ViewMatrixVar = NULL;
g_pd3d10ViewMatrixVar = g_pd3d10Effect
    ->GetVariableByName("ViewMatrix")->AsMatrix();

/* obtain the LightDirection shader variable */
ID3D10EffectVectorVariable* g_pd3d10LightDirectionVectorVar = NULL;
g_pd3d10LightDirectionVectorVar = g_pd3d10Effect
    ->GetVariableByName("LightDirection")->AsVector();

/* obtain the LightColour shader variable */
ID3D10EffectVectorVariable* g_pd3d10LightColourVectorVar = NULL;
g_pd3d10LightColourVectorVar = g_pd3d10Effect
    ->GetVariableByName("LightColour")->AsVector();
```

In summary, the `GetVariableByName` `ID3D10Effect` interface function takes a string value containing the name of the variable declared in the shader/effect program as parameter, returning a pointer to the `ID3D10EffectVariable` interface. The `AsVector` `ID3D10EffectVariable` interface function casts this returned `ID3D10EffectVariable` interface to an `ID3D10EffectVectorVariable` interface so that we can access the vector type. The `AsMatrix` function casts the returned `ID3D10EffectVariable` interface to an `ID3D10EffectMatrixVariable` interface used for reading the shader variable as a matrix type.

Next our renderer sets the values of the shader/effect variables using the `SetMatrix` `ID3D10EffectMatrixVariable` interface for all floating-point matrices and the

`SetFloatVectorArray ID3D10EffectVectorVariable` interface for our four-component floating point vector arrays:

```
g_pd3d10ProjMatrixVar->SetMatrix((float*)&g_ProjectionMatrix);
g_pd3d10WorldMatrixVar->SetMatrix((float*)&g_WorldMatrix);
g_pd3d10ViewMatrixVar->SetMatrix((float*)&g_ViewMatrix );

g_pd3d10LightDirectionVectorVar->SetFloatVectorArray((float*)IncomingLightVector,0, 2);
g_pd3d10LightColourVectorVar->SetFloatVectorArray((float*)IncomingLightColour, 0, 2);
```

The variables declared in the shader program are now set and can be changed during each rendering pass.

Returning to the presented shader program, we declare two structures for the storage of received vertex data and returned pixel data, respectively:

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

```
struct PIXELSHADER_INPUT
{
    float4 Loc : SV_POSITION;
    float3 Norm : TEXCOORD0;
};
```

The first structure, `VERTEXSHADER_INPUT`, holds texture vertex information as received from the Direct3D application. It is used to pass input data to a vertex shader that transforms the input vertex position, defined in object space, to projection space. This is done by multiplying the input vertex position, `IN.Loc`, by a world matrix, thus transforming it from object space to world space. The next transformation multiplies this transformed vertex position, `output.Loc`, with a view matrix, resulting in a world space to view space transformation. The final transformation takes this view space vertex position and multiplies it with a projection matrix to transform the vertex from view space to projection space. The vertex shader also transforms the input vertex normal to world space, finally returning the transformed vertex data via the `PIXELSHADER_INPUT` structure:

```
PIXELSHADER_INPUT LightingVertexShader(VERTEXSHADER_INPUT IN)
{
    PIXELSHADER_INPUT output = (PIXELSHADER_INPUT)0;
```



```
output.Loc = mul(IN.Loc, WorldMatrix);
output.Loc = mul(output.Loc, ViewMatrix);
output.Loc = mul(output.Loc, ProjectionMatrix);

output.Norm = mul(IN.Norm, WorldMatrix);

return output;
}
```

The diffuse lighting on each pixel is determined via a pixel shader. In short, the dot product of the incoming light vector and the surface normal is calculated, with the overall lighting effect determined by multiplying the dot product result by the colour of each light source. All these calculated values are then summed to determine the overall pixel colour.

```
/* pixel shader */
float4 LightingPixelShader(PIXELSHADER_INPUT IN) : SV_Target
{
    float4 finalPixelColour = 0;
    float4 dotPixelColour = 0;

    /* calculate the overall lighting by multiplying the dot product result of the
       incoming light vector and the surface normal with the colour of each light
       source */
    for(int i = 0; i < 2; i++)
    {
        dotPixelColour = dot((float3)LightDirection[i], IN.Norm);
        finalPixelColour += saturate(dotPixelColour * LightColour[i]);
    }

    /* return the overall pixel colour */
    return finalPixelColour;
}
```

The final step is to create the effect technique definition. This effect technique has one rendering pass, P0, specifying the shader states used to perform the lighting operation. It is defined as follows:

```

technique10 LightScene
{
    pass P0
    {
        SetGeometryShader(NULL);
        SetVertexShader(CompileShader(ps_4_0, LightingVertexShader()));
        SetPixelShader(CompileShader(ps_4_0, LightingPixelShader()));
    }
}

```

Returning to the rendering engine's lighting component, all that remains is to create the effect object and technique object that will be used to perform the lighting operation. We can create the effect using this `D3DX10CreateEffectFromFile` function in the following manner:

```

ID3D10Device* g_pd3d10Device = NULL;
ID3D10Effect* g_pd3d10Effect = NULL;

D3DX10CreateEffectFromFile(L"file_name.fx", NULL, NULL,
                          D3D10_SHADER_ENABLE_BACKWARDS_COMPATIBILITY,
                          0, g_pd3d10Device, NULL, NULL,
                          &g_pd3d10Effect, NULL, NULL);

```

Following the effect creation we must obtain the effect technique using the `GetTechniqueByName` `ID3D10Effect` interface function. This function takes a string value containing the name of the technique as parameter, returning a pointer to the `ID3D10EffectTechnique` interface:

```

ID3D10EffectTechnique* g_id3dTechnique = NULL;
g_id3dTechnique = g_id3dEffect->GetTechniqueByName("LightScene");

```

3.4 Reflection and Refraction

The presented rendering engine extends the basic local illumination lighting model through the addition of reflection and refraction effects to result in more realistic and life-like images. When computation processing power is not available, our engine will utilise basic reflective environmental mapping which allows us to simulate complex reflections by mapping real-time computed texture images to the surface of an object (Greene, 1986). Each texture image used for environmental mapping stores a "snapshot" image of the environment surrounding the mapped object. These snapshot images are then

mapped to a geometric object to simulate the object reflecting its surrounding environment (with the cube-map being either calculated on the CPU or GPU, depending on the one most idle). An environment map can be considered an omnidirectional image. Figure 3.6 shows an environmentally mapped object placed within a scene that also makes use of standard environmental mapping to reflect objects in the scene from its “mirror like walls”.



Figure 3.6 An environmentally mapped model and scene. (The most basic form of environmental mapping results in a chrome-like appearance.)

Cube mapping is a type of texturing where six environmental maps are arranged as if they were faces of a cube (Figure 3.7). Images are combined in this manner so that an environment can be reflected in an omnidirectional fashion.

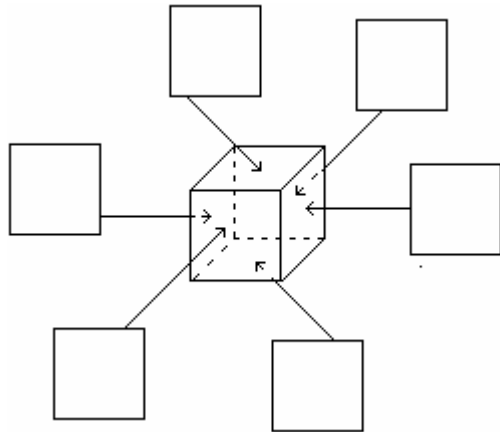
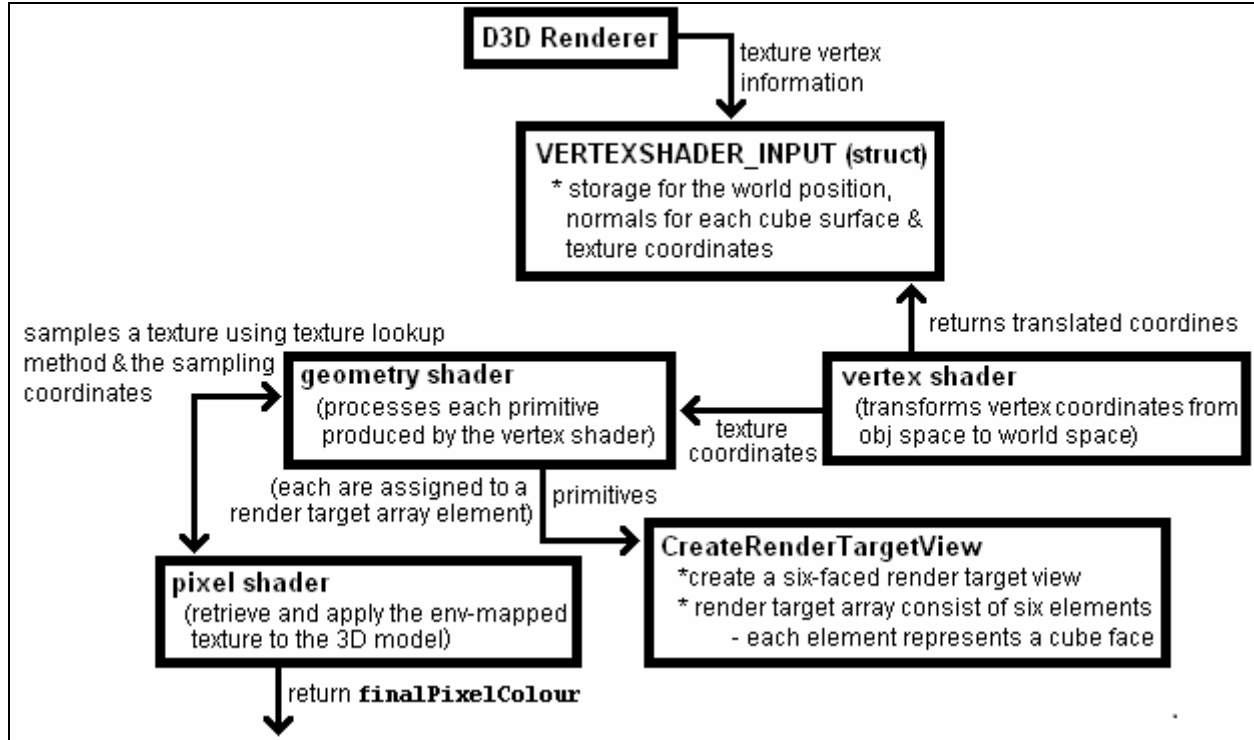


Figure 3.7 A cube map consisting of six texture images.

Cube maps are accessed using a three-dimensional texture coordinate set, specifically a 3D directional vector. We create cube maps by placing a camera at the object's centre and taking 90 degree field-of-view "snapshots" of the environment in each direction of the cube (i.e. along the axes of the Cartesian coordinate system), thus along each of the following: the positive *x-axis*, the negative *x-axis*, the positive *y-axis*, the negative *y-axis*, the positive *z-axis* and the negative *z-axis*.

3.4.1 Implementing Cube Mapping



Cube mapping was, before the advent of shaders, typically implemented in a manual fashion. The conventional process is to acquire snapshots of the environment in each

direction of the cube and subsequently set each of these snapshots as the render target (thus rendering the scene for each side surface of the cube) – the render target view is, in our case, the surface being rendered to with the viewport a window located inside a viewing volume (a semi-infinite, truncated pyramid defined by an image plane window and a near- and far clipping plane). This approach is rather tedious and implementing cube mapping via a vertex and/or pixel shader program greatly improves performance by decreasing the number of rendering passes. One pass is required for each face of the cube when implementing the technique manually. Using Cg, for example (as opposed to our HLSL geometry shader implementation), allows for a vertex/pixel shader approach that can be used with OpenGL programs (or even Direct3D programs not making use of the High Level Shader Language). Direct3D 10 combines an HLSL geometry shader with render target arrays to improve the performance of cube mapping.

Geometry shaders are executed just prior to the rasterizer and stream output pipeline stages. As previously mentioned, these shaders (executing for each primitive) group numerous vertices into a geometric object – thus generating new primitives that can be modified by a pixel shader program. The primitives generated by the geometry shader stage are rasterized into fragments during the pipeline's rasterizer stage.

Our engine (when performing cube mapping on the GPU) uses a geometry shader coupled with a render target array consisting of six elements (each element representing a cube face) to render onto several render targets at the same time. The geometry shader outputs primitives, assigning each output primitive to one of the elements in the render target array.

The `D3D10_RENDER_TARGET_VIEW_DESC` structure is used to describe the render target view (specifically the manner in which a render target resource is interpreted by the pipeline). This structure is defined as follows in the `d3d10.h` header file:

```
typedef struct D3D10_RENDER_TARGET_VIEW_DESC {
    DXGI_FORMAT Format;
    D3D10_RTV_DIMENSION ViewDimension;
    union {
        D3D10_BUFFER_RTV Buffer;
        D3D10_TEX1D_RTV Texture1D;
        D3D10_TEX1D_ARRAY_RTV Texture1DArray;
        D3D10_TEX2D_RTV Texture2D;
        D3D10_TEX2D_ARRAY_RTV Texture2DArray;
        D3D10_TEX2DMS_RTV Texture2DMS;
        D3D10_TEX2DMS_ARRAY_RTV Texture2DMSArray;
        D3D10_TEX3D_RTV Texture3D;
    };
} D3D10_RENDER_TARGET_VIEW_DESC;
```

Its first member, **Format**, describes the resource data format. (It can be set to a constant such as **DXGI_FORMAT_R11G11B10_FLOAT**, representing a 32-bit, three-component floating-point format.) The next member, **ViewDimension**, specifies the manner in which a resource (used in the render-target view) is to be accessed. This member must be set to the same type as that of the defined resource (**D3D10_RTV_DIMENSION_TEXTURE2D** for a 2-D texture, **D3D10_RTV_DIMENSION_TEXTURE2DARRAY** for a 2-D texture array, **D3D10_RTV_DIMENSION_TEXTURE3D** for a 3D texture, etc). The **Buffer** member describes the elements in a buffer resource that will be utilised in a render target view via the specification of two **D3D10_BUFFER_RTV** members, namely, **ElementOffset** (the offset, in byte, from the start of the buffer to the element that will be accessed) and **ElementWidth** (the size, in bytes, of each element stored in the buffer). The next member, **Texture1D**, describes the render target resource as a 1-D texture with **Texture1DArray** specifying the resource as a 1-D texture array. The next two members, **Texture2D** and **Texture2DArray**, specify a 2-D and 2-D array texture, respectively, to use as a render target. Please note, **Texture2DMS** does not specify anything (as multi-sampled 3D textures contain a single sub-resource) while **Texture2DMSArray** specifies the render target resources as a multi-sampled 2-D texture array. The final member, **Texture3D**, specifies the render target as a 3D texture resource.

The **Texture1D** member is declared as a **D3D10_TEX1D_RTV** structure and has one member, namely, **MipSlice**. It specifies the mipmap level to use in a render target view. (A mipmap is a series of pre-filtered texture images of varying resolution levels; '0' indicates the first level. When using mipmaps, Direct3D automatically maps a suitable texture, based on size in pixels, to the object being mapped.) The **Texture1DArray** member of type **D3D10_TEX1D_ARRAY_RTV** shares its first member, **MipSlice** with the **D3D10_TEX1D_RTV** structure. It has two additional members, namely **FirstArraySlice** (specifying the texture array's first texture that will be used in the render target view) and **ArraySize** (specifying the number of textures that can be used in the render target view). The **D3D10_TEX2D_RTV Texture2D** member has one member, **MipSlice**, specifying the mipmap level to use in a render target view. The **Texture2DArray D3D10_TEX2D_ARRAY_RTV** member has three members to specify the mipmap levels and textures to use in a render target view, namely, **MipSlice**, **FirstArraySlice** and **ArraySize**. Of the remaining three members, **Texture2DMS** does not have any members to specify since multi-sampled two-dimensional textures contain only one sub-resource. The **Texture2DMSArray** member of type **D3D10_TEX2D_ARRAY_RTV** has two members, **FirstArraySlice** and **ArraySize** – both with the same function as their previously discussed counterparts. The final member, **Texture3D**, of type **D3D10_TEX3D_RTV** has the following members: **MipSlice**, **FirstWSlice** (defining the first depth level that will be used by the render target view) and **WSize** (specifying the number of depth levels).

We can now define a six-faced render target view using this `D3D10_RENDER_TARGET_VIEW_DESC` structure as follows:

```
/* define the render target view description structure */
D3D10_RENDER_TARGET_VIEW_DESC renderTargetViewDescription;

/* set renderTargetViewDescription's Format member */
renderTargetViewDescription.Format = textureDescription.Format;

/* set renderTargetViewDescription's ViewDimension member */
renderTargetViewDescription.ViewDimension = D3D10_RTV_DIMENSION_TEXTURE2DARRAY;

/* set the resource type as a 2-D texture array (Texture2DArray), subsequently setting
   its members to represent an array of 6 render targets (one for each face of the
   cube) */
renderTargetViewDescription.Texture2DArray.MipSlice = 0;
renderTargetViewDescription.Texture2DArray.FirstArraySlice = 0;
renderTargetViewDescription.Texture2DArray.ArraySize = 6;
```

Next the `CreateRenderTargetView` `ID3D10Device` interface function is called to create a render target view that will be used to access data in the defined resource. This function is declared as follows in the `d3d10.h` header file:

```
HRESULT CreateRenderTargetView(
    ID3D10Resource *pResource,
    const D3D10_RENDER_TARGET_VIEW_DESC *pDesc,
    ID3D10RenderTargetView **ppRTView
);
```

Its first parameter, `pResource`, is a pointer to either a buffer resource such as a vertex buffer, index buffer or a shader constant buffer or alternatively a texture resource (as in our case). The second parameter, `pDesc`, takes a pointer to the render target view description structure, `D3D10_RENDER_TARGET_VIEW_DESC`. Its last parameter, `ppSRView`, takes the address of a pointer to the render target view interface, `ID3D10RenderTargetView`, dealing with how the pipeline outputs data during the rendering process. The following code sample creates a render target view so that the cube texture can be rendered:

```
/* declare a 2-D texture interface to manage texel data */
ID3D10Texture2D* g_pEnvironmentalMap;

/* declare a ID3D10RenderTargetView interface */
```



```
ID3D10RenderTargetView* g_pEnvironmentalMapRenderTargetView;

/* create the render target resource view */
HRESULT_ = g_id3DDevice->CreateRenderTargetView (g_pEnvironmentalMap,
                                                &renderTargetViewDescription,
                                                &g_pEnvironmentalMapRenderTargetView);
```

The six faces of the cube are rendered at the same time by setting the render target view as active when rendering onto the cube map. This is done by calling the `OMSetRenderTargets` `ID3D10Device` interface function. This function binds the render target view to the pipeline so that the pipeline's output can be written onto the back buffer. The `OMSetRenderTargets` interface method takes three parameters, namely, the number of render targets to bind to the pipeline, a pointer to the `ID3D10RenderTargetView` interface and a pointer to the depth-stencil view:

```
/* define an array of render target views */
ID3D10RenderTargetView* arrayRenderTargetViews[1] =
    {g_pEnvironmentalMapRenderTargetView};

/* define a depth-stencil view for controlling the texture resource utilised during
   the depth-stencil test - specifically the Depth stencil view of the environment map
   for all six faces */
ID3D10DepthStencilView* pDepthStencilView;

g_id3DDevice->OMSetRenderTargets( sizeof(arrayRenderTargetViews)/
                                sizeof(arrayRenderTargetViews[0]),
                                arrayRenderTargetViews,
                                pDepthStencilView);
```

We render the scene onto the current render target (the surface being rendered to) by first clearing the render target, then clearing the depth stencil buffer, followed by the setup of the appropriate matrices and drawing of the actual object (for example, a 3D mesh) that is to be cube mapped. The scene is then rendered onto the cube texture (by first saving the old viewport and then specifying the new viewport for rendering to the cube map and computing the view matrices used for this rendering – the eye coordinates are set at the centre of the cube mapped object after we have combined the six different view directions to obtain the final view matrix). Following this, we render one cube face at a time, restoring the saved viewport and rendering the final reflective scene.

The actual cube mapping is done via a geometry shader. This geometry shader is used to output each primitive (points, lines, polygons) to every render target – six surfaces in total. The cube mapping effect also uses a vertex shader to transform vertex coordinates

from object space (the initial position and orientation of objects before any transformation is applied) to world space – coordinate space where we have a reference to the viewer's position (as required for the geometry shader to function). We will now look at this vertex shader used for propagating texture coordinates from the application program to the geometric shader.

The first step is to declare a vertex shader storage structure to store the world position, normals for each cube surface and texture coordinates:

```
struct VERTEXSHADER_CUBEMAP
{
    float4 Loc : POSITION;
    float2 Tex : TEXCOORD0;
    float3 Normals[6] : SIXNORMS;
};
```

Next the vertex shader is defined to transform vertex coordinates from object space to world space. This shader returns these translated coordinates and forwards the texture coordinates:

```
VS_OUTPUT_CUBEMAP CubemapVertexShader(float4 Loc: POSITION, float3 Normal : NORMAL,
                                       float2 Tex : TEXCOORD)
{
    /* declare a VERTEXSHADER_CUBEMAP structure */
    VERTEXSHADER_CUBEMAP output;

    /* transform vertex positions from object space to world space */
    output.Loc = mul(Loc, worldProjection);

    /* pass the texture coordinates to the geometric shader */
    output.Tex = Tex;

    return output;
}
```

The implemented geometric shader processes each primitive produced by the above defined vertex shader. It does this by looping through all the cube faces/cube maps and for each face, looping an additional three times to create the vertices making up a triangle. The geometric shader calculates the position of the output vertices used by the rasterizer to rasterize the triangle – i.e. assigning a primitive to each distinct render target in the render target array. The geometric shader also transforms the world space vertices using view transformations for every render target view per iteration.

Specifying the geometric shader, we start by creating a structure to store the projection coordinates, texture coordinates and render target array index used for controlling the render target to which a primitive is written (using the `SV_RenderTargetArrayIndex` HLSL semantic):

```
struct GEOMETRYSHADER_CUBEMAP
{
    /* projection coordinates */
    float4 Loc : SV_POSITION;

    /* texture coordinates */
    float2 Tex : TEXCOORD0;

    /* the index specifying the render target to which the primitive is written */
    int RenderTargetArrayIndex : SV_RenderTargetArrayIndex;
};
```

Following this structure we create the actual geometry shader:

```
/* declare a view matrix for the cube map */
matrix g_mCubemapViewMatrix[6];

/* declare a projection matrix for the cube map */
matrix projectionMatrix : PROJECTION;

/*the geometry shader */
[maxvertexcount(24)]
CubemapGeometryShader(triangle VERTEXSHADER_CUBEMAP Input[3],
                      inout TriangleStream<GEOMETRYSHADER_CUBEMAP> GS_Output)
{
    for(int i = 1; i <= 6; i++)
    {
        /* declare a GEOMETRYSHADER_CUBEMAP structure */
        GEOMETRYSHADER_CUBEMAP output;

        /* set the render target array's index */
        output.RenderTargetArrayIndex = i;

        /* compute the screen vertex & texture coordinates */
        for(int j = 1; j <= 3; i++)
        {
            output.Loc = mul(Input[j].Loc,g_mCubemapViewMatrix[i]);
            output.Loc = mul(output.Loc, projectionMatrix);
```



```
        output.Tex = Input[j].Tex;
        GS_Output.Append(output);
    }
    GS_Output.RestartStrip();
}
}
```

The first parameter, `maxvertexcount`, is set to 24 – hence limiting the maximum number of vertices that the shader can output at a time to this value. Two interesting HLSL stream object functions are used, namely `Append` and `RestartStrip`. `Append` adds geometry shader data to an output stream by appending it to the data already in the output stream. `RestartStrip` terminates the current primitive strip, in this case a triangle strip, signalling the start of a new strip. This geometry shader writes one triangle to each render target texture (the six faces of the cube) in one rendering pass.

Following this geometry shader definition we create a pixel shader to retrieve and apply the environmentally mapped texture to the 3D model.

The first step is to define the sampling method which will control the texture lookup method:

```
SamplerState samplingMethod
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};
```

We define the pixel shader, retrieving and applying the environmental texture to the object, as follows:

```
/* declared 2-D texture variable */
Texture2D g_texture;

float4 CubemapPixelShader(GEOMETRYSHADER_CUBEMAP inputcoords):SV_Target
{
    /* samples a texture using the specified texture lookup method and a floating-
       point value, inputcoords.Tex, specifying the sampling coordinates */
    return g_texture.Sample(samplingMethod, inputcoords.Tex);
}
```

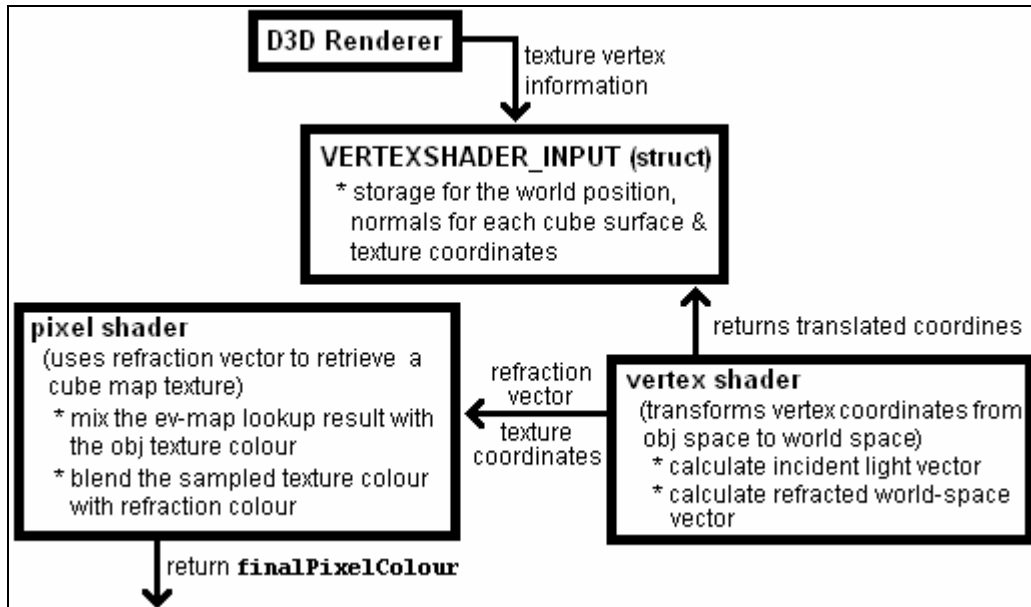
We can now specify the effect technique that will set the previously defined vertex, pixel and geometry shaders. This effect technique has one rendering pass, namely `P0`:

```
technique10 RenderCubemap
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, CubemapVertexShader()));
        SetGeometryShader(CompileShader(gs_4_0, CubemapGeometryShader()));
        SetPixelShader(CompileShader(ps_4_0, CubemapPixelShader()));
    }
};
```

The final step is to render the reflecting mesh.

The presented rendering engine, as outlined in the next two sections, further supports refractive environmental mapping, the Fresnel effect and chromatic dispersion resulting in an object's colour being blended with reflections from its cube map. Thus, when the processing power is available, basic cube-mapped reflections, as just discussed, can be extended to appear more lifelike. To accomplish this, we basically have to write shaders to approximate the Fresnel reflection function and chromatic dispersion so that the object colour is blended with reflections from the cube map. The *Fresnel effect* combines reflection and refraction, i.e. allowing us to simulate the accurate reflection off and refraction through a surface using a number of Fresnel equations. *Chromatic dispersion* extends the basic refraction model to consider the wavelength of the incoming light, that is, to recognise that certain light colours are refracted more than others. Specifically, the higher the wavelength of the colour, the more is it refracted. For example, green has a wavelength ranging from 495 to 570 nm with orange ranging from 590 to 630 nm. The colour orange will thus refract more than green due to its higher wavelength. Sections 3.4.2 and 3.4.3 deal with these advanced techniques as utilised by our rendering engine.

3.4.2 Implementing Basic Refraction



With the cube mapping technique discussed in section 3.4.1, we are able to simulate basic environmental reflections. Environmental mapping, as presented, results in the chrome-like appearance of objects (see Figure 3.6). The main reason for this chrome-like appearance is our technique’s failure, as an approximation, to blend an object’s colour with the reflections from the cube map – in short, a failure to consider the effect of refraction. Our previous model will now be extended to incorporate refraction.

Refraction is the change in direction of a light ray due to a variance in material density (for example, a light wave travelling from air into water). This directional change is the result of a light ray’s speed. For example, light travels faster in air than in water – hence, light travels more slowly in denser materials causing a change in direction where the light enters this material. Figure 3.8 shows the refraction of light rays in water.

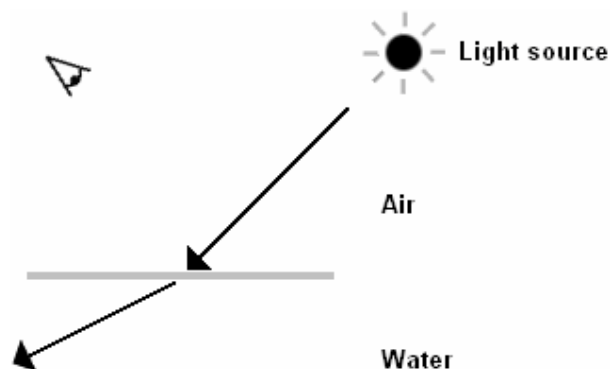


Figure 3.8 Refraction due to light passing from a lower- to a higher density material.

Snell's Law, also known as Descartes' law, is used to calculate the degree of refraction at the boundary of a lower- and higher density material. This law describes the correlation between the incoming light direction and the amount of refraction based on the index of refraction for each material. The *index of refraction* is simply a measure based on the manner in which the material affects the speed of light – the higher the index of refraction, the slower the speed of light. Common indices of refraction are 1.0 for a vacuum, 1.0003 for air, 1.333 for water and 1.5 for glass. Snell's Law (illustrated in Figure 3.9) can be mathematically expressed as follows:

$n_1 \sin \theta_1 = n_2 \sin \theta_2$, with n_1 the refraction index of the lower density material, n_2 the refraction index of the higher density material, θ_1 the incident angle and θ_2 the angle of refraction.

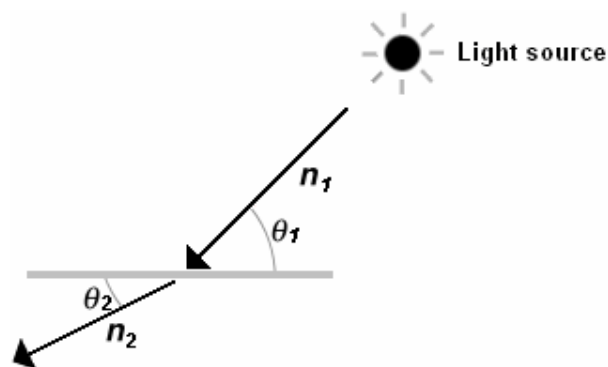


Figure 3.9 Snell's Law

Adding refraction to an environmental map involves tracing each incident ray from the point of view to the surface of the object. In the case of reflection, this ray bounces off the surface of the object. In the case of refraction, this ray changes direction inside the object. The shader implementation for refraction is thus very similar to the environmental mapping implementation of section 3.1.3.1.

When implementing refraction, we only consider one refraction ray per incoming ray of light as opposed to multiple refractions (as the case with real-life where refraction also occurs at the exit boundary of the object). Refraction is thus only simulated to a certain degree. However, refraction is so complex that the human eye will experience significant difficulty in identifying such minor faults with the resulting rendering.

We now present our shader implementation for refraction. This sample utilises the `refract` intrinsic function to calculate the refraction vector. It takes an incident vector – a light ray as first parameter, and a surface normal as second parameter. Its third parameter is the index of refraction – the ratio of indices of refraction of the two materials. It returns the refraction vector. The refracted vector has the same magnitude as the incident ray.

A vertex shader program is used to perform the per-vertex refraction calculations.

The first step is to specify the name of the vertex program's entry function, `main_vertex` in our example:

```
void main_vertex(float4 objectspaceVertexPosition : POSITION,
                float3 objectspaceVertexNormal : NORMAL,
                float2 inputTextureCoordinates : TEXCOORD0,

                out float4 outputVertexPosition : POSITION,
                out float2 outputTextureCoordinates : TEXCOORD0,
                out float3 refractionVector : TEXCOORD1,

                /* parameters supplied by the application program */
                uniform float3 pointOfView,
                uniform float3 refractionRatio
                uniform float4x4 modelToWorldTransformation,
                uniform float4x4 modelviewProjection)
{
```

The first step is to calculate the clip-space position (as mandatory for all vertex programs):

```
/* transform the vertex position into homogeneous clip-space coordinates */
outputVertexPosition = mul(modelviewProjection, objectspaceVertexPosition);
```

Next the input texture coordinates are assigned to the output texture coordinates:

```
/* assign the input texture coordinates to the output texture coordinates */
outputTextureCoordinates = inputTextureCoordinates;
```

As with reflection and environmental maps, refraction is defined in terms of world space coordinates. We must thus transform the normal and vertex position from object space to world space by multiplying both of them by the `modelToWorldTransformation` matrix. This transformation is required since we wish to calculate the refraction vector in terms of world space coordinates. This transformation is done as follows:

```
/* transform the vertex position and normal to world space coordinates */
float3 worldspaceVertexPosition = mul(modelToWorldTransformation,
                                     objectspaceVertexPosition);
float3 worldspaceVertexNormal = mul(modelToWorldTransformation,
                                    objectspaceVertexNormal);
```

```
/* normalise the vertex normal */
worldspaceVertexNormal = normalize(worldspaceVertexNormal);
```

The final operation is to calculate both the incident and refraction vectors. The incident vector is the vector traced from the point-of-view to the vertex. The incident vector is calculated using simple subtraction:

```
/* calculate the incident light vector */
float3 incidentVector = worldspaceVertexPosition - pointOfView;
```

Using the ratio of refraction and the incident and vertex normal, we can calculate the refracted world-space vector:

```
/* calculate the refraction vector */
float3 refractionVector = refract(incidentVector, worldspaceVertexNormal,
                                refractionRatio);
}
```

We now define a pixel shader program that uses this refraction vector to retrieve a cube map texture – the environmental map. This time we extend our previous pixel shader to mix the environment map lookup result with the object's texture colour. This is done by performing a texture lookup of the object's current colour, blending the sampled texture colour with the refraction colour – thus resulting in a much more realistic looking object (due to no material being a perfect refractor). Our original environmental mapping shader is extended in a similar fashion, in its case blending the sampled texture colour with the reflection colour instead of the refraction colour.

The first step is to specify the name of the fragment program's entry function, `main_fragment` in our sample. It has the following signature:

```
void main_fragment(float3 refractionVector : TEXCOORD0,
                  float2 inputTextureCoordinates : TEXCOORD1,
                  out float4 outputColour : COLOR,
                  /* parameter supplied by the application program */
                  uniform samplerCUBE environmentMap,
                  uniform sampler2D lookupTextureColour)
```

Within the body of `main_fragment`, the interpolated refraction vector is used to determine the environment map's refracted colour. We use the `texCUBE` texture lookup function to

do this. This function takes two parameters; a cube map and a three component texture coordinate set – the refraction vector:

```
/* obtain the refracted colour */  
float4 refractionColour = texCUBE(environmentMap, refractionVector);
```

Next we perform a texture colour lookup using the `tex2D` function – this function performs a 2D texture lookup determining the fragment's colour (the '2D' suffix indicating the sampling of 2D sampler objects). It takes two parameters, the first being a sampler object and the second a texture coordinate set specifying the location to sample the object at. This function produces sampled data as output which is returned by the fragment program through the colour variable):

```
float4 textureColour = tex2D(lookupTextureColour, inputTextureCoordinates);
```

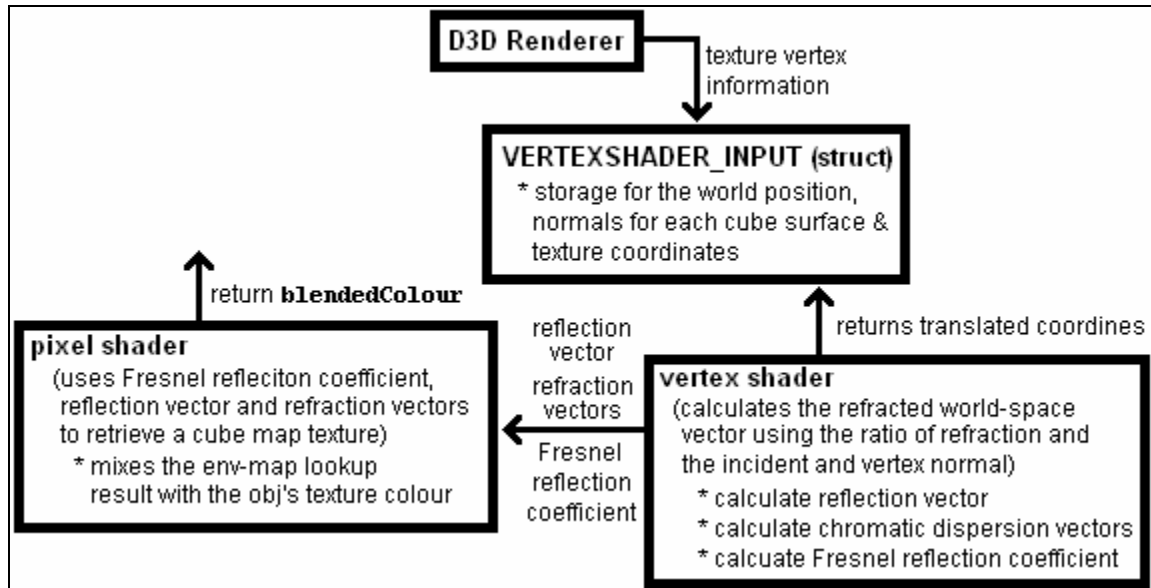
Following this, the sampled texture colour is blended with the refraction colour using the `lerp` function. This function performs a linear interpolation, computing the average of two colour samples. Its first two parameters are the colour vectors to average, with its third parameter controlling the amount of averaging, for example, a weight of '0.5' resulting in uniform averaging. Setting this weight to '0' results in no reflection or refraction. Conversely, setting the weight to '1' will lead to the program not considering the texture colour, thus producing a completely reflective or refractive object:

```
float4 blendedColour = lerp(textureColour, refractionColour, 0.5);
```

Finally, this linearly interpolated blended colour is assigned to the output colour:

```
/* set the refracted colour */  
colour = blendedColour;
```

3.4.3 Reflection and Refraction Extended



We will now further extend our previous implementation using a number of advancements to improve the overall reflection effect, thus resulting in even more realistic and lifelike images.

Reflection, as mentioned, is the change in direction of a light ray where the light ray is reflected back into the originating material upon contact with the surface of another material. Perfect reflection is characterised by the angle of incidence, θ_1 , being equal to the angle of reflection, θ_2 . Figure 3.10 shows the perfect reflection of light.

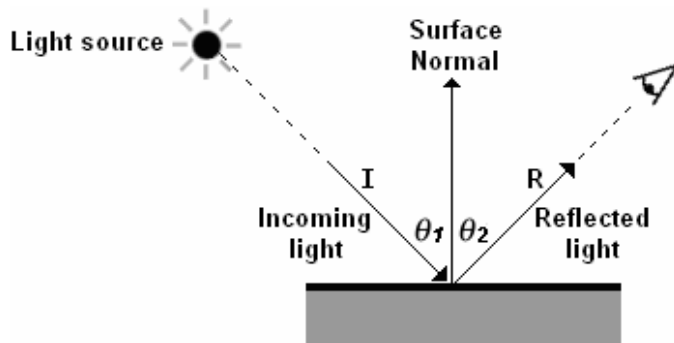


Figure 3.10 Perfect reflection ($\theta_1 = \theta_2$).

We can subsequently compute the reflection vector, R , by taking the incident vector, I , and subtracting two times the surface normal, N , multiplied by the dot product between the surface normal and the incident light:

$$R = I - 2N(N \cdot I).$$

Our shader programs, as mentioned, utilise the `refract` Standard Library Function to calculate the refraction vector. This function takes an incident vector – a light ray as first parameter, and a surface normal as second parameter, subsequently returning the reflection vector of the incident ray. (The incident light wave is partially refracted and partially reflected.)

The Fresnel effect was previously defined as a series of equations combining reflection and refraction to accurately simulate the reflection off and refraction through a surface. These equations are used to determine the amount of light reflected and refracted. However, using these equations directly is a bit excessive and we rather approximate the Fresnel equations into the equation `fresnel bias + fresnel scale * pow(1 + dot(incident ray, surface normal), fresnel power)` that can easily be incorporated into the previously presented reflection shader programs:

```
FresnelReflectionCoefficient = 0.183673 + 0.816327 * pow(1.0 - dot(incidentVector,
                                                                    worldspaceVertexNormal), 5.0);
```

This equation is based on the principle of Fresnel reflection; namely, that when the incident vector is parallel to the surface normal, then the majority of light is refracted with the reflection coefficient approaching '0' (Fernando and Kilgard, 2003). As the angle between the incident vector and surface normal increases, so does the amount of light reflected. This Fresnel reflection coefficient is used in the calculation of the final colour contribution resulting from both reflection and refraction. The Fresnel reflection coefficient is simply used as the `lerp` function's weight.

Chromatic dispersion can be defined as an extension to the basic lighting model that deals with the fact that certain light colours are refracted more than others. Chromatic dispersion models the refraction of red, green and blue light. We can thus extend the single refracted ray lookup (as done previously) by using these refracted light rays for our environmental map lookup. Adding chromatic dispersion to our current reflection and refraction models result in the rainbow-like refraction of light – as the case with the dispersion of a light beam in a prism (Figure 3.11).

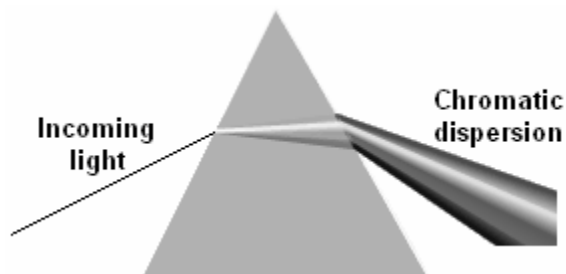


Figure 3.11 Chromatic dispersion of light.

Our basic shader calculates the refracted world-space vector using the ratio of refraction and the incident and vertex normal:

```
/* calculate the refraction vector */  
float3 refractionVector = refract(incidentVector, worldspaceVertexNormal,  
                                refractionRatio);
```

Incorporating chromatic dispersion into our program requires three refraction vectors, one for each of the primary colours:

```
float3 refractionVectorRed = refract(incidentVector, worldspaceVertexNormal,  
                                    refractionRatioRed);  
  
float3 refractionVectorBlue = refract(incidentVector, worldspaceVertexNormal,  
                                       refractionRatioBlue);  
  
float3 refractionVectorGreen = refract(incidentVector, worldspaceVertexNormal,  
                                       refractionRatioGreen);
```

When the computational processing power is available, we extend our previous reflection implementation (using `reflect`, the library function) to incorporate a texture lookup of the object's current colour, blending the sampled texture colour with the reflection colour – thus resulting in a much more realistic looking object. The previous shader is further extended to incorporate chromatic dispersion and the Fresnel effect. This program utilises the `reflect` and `refract` library functions to calculate the reflection vector and refraction vectors, respectively.

A vertex shader program is used to calculate the reflection vector together with the chromatic dispersion vectors and the Fresnel reflection coefficient which will be sent to a fragment shader.

The first step is to specify the name of the vertex program's entry function, `main_vertex` that has the following signature:

```
void main_vertex(float4 objectspaceVertexPosition : POSITION,  
                float3 objectspaceVertexNormal : NORMAL,  
                float2 inputTextureCoordinates : TEXCOORD0,  
  
                out float fresnelReflectionCoefficient: COLOR,  
                out float4 outputVertexPosition : POSITION,  
                out float3 reflectionVector : TEXCOORD0,  
                out float3 refractionVectorRed : TEXCOORD1,  
                out float3 refractionVectorBlue : TEXCOORD2,
```

```

out float3 refractionVectorGreen : TEXCOORD3,

/* parameters supplied by the application program */
uniform float3 pointOfView,
uniform float4x4 modelToWorldTransformation,
uniform float4x4 modelviewProjection,
uniform float3 refractionRatioRed,
uniform float3 refractionRatioBlue,
uniform float3 refractionRatioGreen)

```

In the body of this function, we start by calculating the clip-space position:

```

/* transform the vertex position into homogeneous clip-space coordinates */
outputVertexPosition = mul(modelviewProjection, objectspaceVertexPosition);

```

Next the input texture coordinates are assigned to the output texture coordinates:

```

/* assign the input texture coordinates to the output texture coordinates */
outputTextureCoordinates = inputTextureCoordinates;

```

Reflection and environmental maps are defined in terms of world space coordinates. We must thus, as discussed previously, transform the normal and vertex position from object space to world space by multiplying both of them by the `modelToWorldTransformation` matrix. This transformation is required since we wish to calculate the reflection vector in terms of world space coordinates. This transformation is done as follows:

```

/* transform the vertex position and normal to world space coordinates */
float3 worldspaceVertexPosition = mul(modelToWorldTransformation,
                                     objectspaceVertexPosition);
float3 worldspaceVertexNormal = mul(modelToWorldTransformation,
                                    objectspaceVertexNormal);

/* normalise the vertex normal */
worldspaceVertexNormal = normalize(worldspaceVertexNormal);

```

The final operation is to calculate both the incident and reflection vectors. The incident vector is the vector traced from the point-of-view to the vertex. The incident vector is calculated using simple subtraction:

```

/* calculate the incident light vector */
float3 incidentVector = worldspaceVertexPosition - pointOfView;

```


Using the incident and vertex normal, we can calculate the reflected world-space vector:

```
/* calculate the reflection vector */  
float3 reflectionVector = reflect(incidentVector, worldspaceVertexNormal);  
  
/* normalise the incident Vector */  
incidentVector = normalize(incidentVector);
```

The next step is to calculate the Fresnel reflection coefficient via our previously listed approximation:

```
fresnelReflectionCoefficient = 0.183673 + 0.816327 *  
    pow(1.0 - dot(incidentVector,  
    worldspaceVertexNormal), 5.0);
```

We lastly calculate the chromatic dispersion refraction vectors (one for each of the primary colours):

```
float3 refractionVectorRed = refract(incidentVector, worldspaceVertexNormal,  
    refractionRatioRed);  
  
float3 refractionVectorBlue = refract(incidentVector, worldspaceVertexNormal,  
    refractionRatioBlue);  
  
float3 refractionVectorGreen = refract(incidentVector, worldspaceVertexNormal,  
    refractionRatioGreen);
```

We now define a fragment shader program that uses the calculated Fresnel reflection coefficient, reflection vector and refraction vectors to retrieve a cube map texture – the environmental map. Our shader also mixes the environment map lookup result with the object's texture colour. This is done via a texture lookup of the object's current colour and the blending of this sampled texture colour with the reflection and refraction colours – thus resulting in a highly accurate lighting model.

The first step is to specify the name of the fragment program's entry function, `main_fragment` in our sample:

```
void main_fragment(float3 reflectionVector : TEXCOORD0,  
    out float3 refractionVectorRed : TEXCOORD1,  
    out float3 refractionVectorBlue : TEXCOORD2,  
    out float3 refractionVectorGreen : TEXCOORD3,  
    float fresnelReflectionCoefficient: COLOR,
```

```
out float4 outputColour : COLOR,  
  
/* parameter supplied by the application program */  
uniform samplerCUBE environmentMap)  
{
```

The fragment program uses the interpolated reflection and refraction vectors to determine the environment map's reflected colour. We use the `texCUBE` texture lookup function to do this. This function takes two parameters; a cube map and a three component texture coordinate set – the reflection vector:

```
/* obtain the reflection colour */  
float4 reflectionColour = texCUBE(environmentMap, reflectionVector);  
  
/* obtain the refraction colour */  
float4 refractionColour.r = texCUBE(environmentMap, refractionVectorRed).r;  
float4 refractionColour.b = texCUBE(environmentMap, refractionVectorBlue).b;  
float4 refractionColour.g = texCUBE(environmentMap, refractionVectorGreen).g;
```

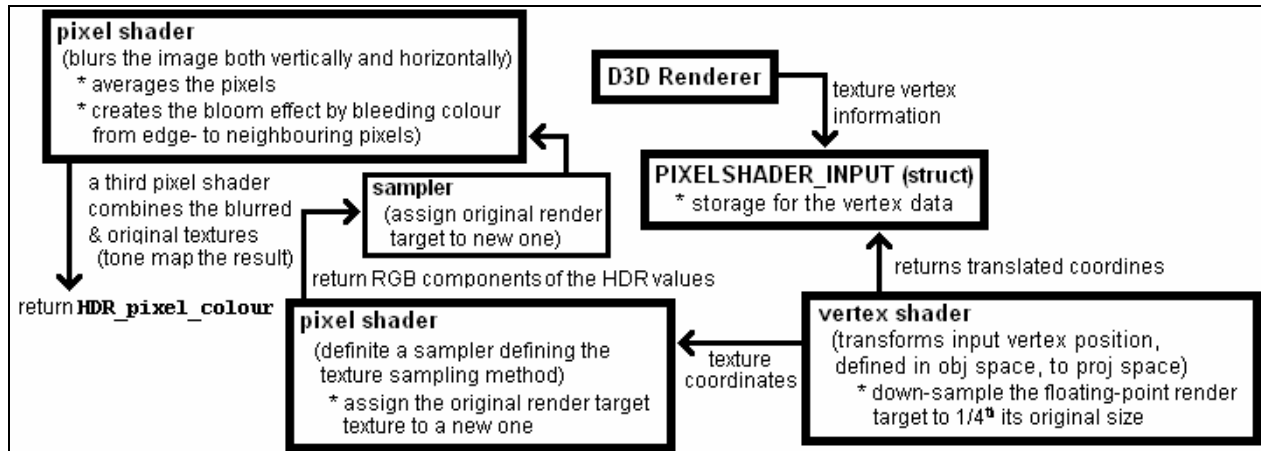
Following this, we blend the sampled refraction texture colour with the reflection colour using the `lerp` function. The refraction colour's weight, given as the function's third parameter, is set to the calculated Fresnel reflection coefficient:

```
float4 blendedColour = lerp(textureColour,  
                           reflectionColour,  
                           fresnelReflectionCoefficient);
```

We finally assign this linearly interpolated blended colour to the output colour:

```
/* set the blended colour */  
colour = blendedColour;
```

3.5 Adding High Dynamic Range (HDR) Lighting



High dynamic range lighting, also known as high dynamic range rendering (HDRR), is the rendering of lighting using more than 256 colour shades for each of the primary colours, namely, the red, green and blue components. Thus, we can, thanks to this technique, use 16 to 32-bit colours per RGB channel as opposed to the normal 8 – eliminating luminance and pixel intensity being clamped to a [0, 1] range. This allows our engine the display of light sources over 100 000 times brighter than normally possible.

HDR's wide colour range leads to the effect of bright lights appearing very bright, with dark areas looking even darker at the same time. HDR lighting results in the full visibility of both very dark and fully lit areas; unlike normal lighting, or *low dynamic range lighting*, where details are hidden in dark scenes when contrasted by a fully lit area. Using this form of lighting generally leads to a more vibrant looking scene. Figure 3.12 shows an example of HDR from Valve Software's *Half-Life 2: Lost Coast* technology showcase.



Figure 3.12 High dynamic range lighting.

HDR lighting generally makes use of two techniques, namely, tone mapping and the bloom effect. *Tone mapping* is used to approximate real-world luminance, which has an extremely high dynamic range, to a computer monitor which has a limited range of luminance values. The *bloom effect* basically blends light sources beyond their natural edges – causing the edges of a bright light source to overlap nearby geometry, thus creating the illusion of an even brighter light.

Floating-point textures are normally used for the storage of HDR lighting colour information. This colour data can also be encoded using integer textures as discussed by the DirectX 10 SDK's "HDRFormats10" sample. The main reason for using floating-point textures is due to the pixel shader clamping integer textures to the range [0, 1]. Floating-point textures are not clamped at all and can thus contain a wide range of values.

The following steps outline the process of rendering a scene using HDR lighting:

- 1) Load the HDR floating-point values into a buffer (a floating-point render target).
- 2) Apply the Bloom effect.
 - a. Down-sample the buffer to 1/4th its original size. This is required so that the bloom effect is only ranged from edge pixels to neighbouring ones.
 - b. Blur the image both vertically and horizontally (thus averaging the pixels and consequently creating the bloom effect by bleeding colour from edge- to neighbouring pixels).
- 3) Combine the blurred and original texture.
- 4) Tone map the composed texture.

We start by reading the red, green and blue components of our HDR floating-point texture – such as images stored in the radiance HDR (".hdr" or ".pic") file format. This image will be used to texture a simple quadrilateral. This quadrilateral will in turn be illuminated using high dynamic range lighting. The RGB components of our HDR floating-point texture are stored as an array of floating-point values. These RGB floating-point values are in turn set to a floating-point render target.

Our first HLSL shader is used to down-sample the floating-point render target to 1/4th its original size. We start by declaring the pixel offset as used in our vertex shader, also declaring a structure for the storage of vertex data:

```
/* pixel offset = 1 / 1280 and 1 / 1024 */  
float2 GlobalPixelOffset = float2(0.00078125, 0.000976562);
```

```
struct PIXELSHADER_INPUT
```

```
{
    float4 Loc: POSITION;
    float2 Texture: TEXCOORD0;
};
```

The associated vertex shader starts by transforming the input vertex position, defined in object space, to projection space. This is done by multiplying the input vertex position, `IN.Loc` by a world matrix. The next transformation multiplies this transformed vertex position, `output.Loc` by a view matrix, resulting in a world space to view space transformation. The final transformation takes this view space vertex position and multiplies it by a projection matrix to transform the vertex from view space to projection space. The shader's final routine outputs the texture coordinates:

```
/* vertex shader */
PIXELSHADER_INPUT DownSamplerVertexShader(float3 IN: POSITION,
                                           float2 IN_TEXTURE: TEXCOORD0)
{
    PIXELSHADER_INPUT output;

    /* transforms the input vertex position */
    output.Loc = mul(IN.Loc, WorldMatrix);
    output.Loc = mul(output.Loc, ViewMatrix);
    output.Loc = mul(output.Loc, ProjectionMatrix);

    output.Texture = IN_TEXTURE + (GlobalPixelOffset/2);

    return output;
}
```

In the case of our pixel shader we define a sampler (an external object that can be sampled, such as a texture) specifying the manner in which the texture will be sampled. We simply assign the original render target texture (stored in the buffer where the video card draws pixels for a scene that is being rendered) to a new one:

```
texture sampledTexture;

SamplerState samplingMethod
{
    Texture = sampledTexture;
};
```

The RGB components of the HDR values are only returned if they are in fact HDR values, thus ignoring all low dynamic range lighting values – the function `OnlyHDR`, used by the pixel shader, is declared as follows:

```
float4 OnlyHDR(float4 colour)
{
    if(colour.r > 1.0f && colour.g > 1.0f && colour.b > 1.0f)
    {
        return colour;
    }
    else
        float4 new_colour = {0.0f, 0.0f, 0.0f, 0.0f};
        return new_colour;
}
```

The pixel shader performs a texture colour lookup using the `tex2D` function, subsequently rendering this texture onto the resized render target, finally outputting the RGB components of the HDR values only if they are in fact HDR values:

```
float4 DownSamplerPixelShader(float2 IN_TEXTURE: TEXCOORD0) : COLOR0
{
    float4 colour = tex2D(samplingMethod, IN_TEXTURE);
    float4 sampledColour = OnlyHDR(colour);
    return sampledColour;
}
```

Following this we need to blur the image both vertically and horizontally (thus averaging the pixels and consequently creating the bloom effect by bleeding colour from edge- to neighbouring pixels). This is done using a simple Gaussian effect, the result of which is shown in Figure 3.13.

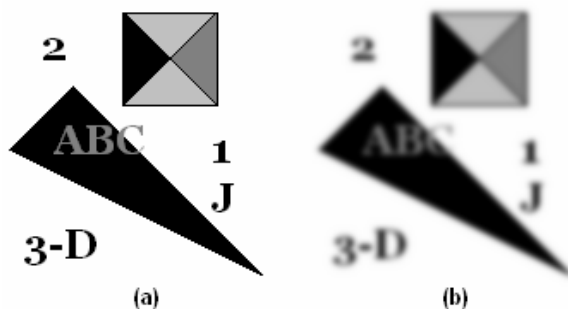


Figure 3.13 (b) Gaussian blur of (a), a simple image.

We start by declaring a sampler assigning the original render target texture to a new one:

```
texture blurredTexture;

SamplerState blurredSampler
{
    Texture = blurredTexture;
};
```

A texture sampler for the original non-blurred texture, `originalTexture`, is also declared:

```
Texture2D originalTexture;

SamplerState originalSampler
{
    Texture = originalTexture;
};
```

Next a pixel shader to do the actual Gaussian blur is defined. The `xoffset` variable is the texel width and the `yoffset` variable the texel height. For example, an image of 256 by 512 pixels will have a `xoffset` of $1/256 = 0.00390625$ and a `yoffset` of $1/512 = 0.001953195$:

```
float4 GaussianBlurPixelShader(float2 IN_TEXTURE: IN_TEXTURE) : COLOR0
{
    float4 colour = tex2D(blurredSampler, IN_TEXTURE);

    /* sample eight pixels at each side */
    for(int pixel_number = 1; pixel_number <= 8; pixel_number++)
    {
        /* blur in the x-axis direction */
        colour += tex2D(blurredSampler, IN_TEXTURE + (XOffset* pixel_number)) *
            GaussianWeights[pixel_number];

        colour += tex2D(blurredSampler, IN_TEXTURE - (XOffset * pixel_number)) *
            GaussianWeights[pixel_number];

        /* blur in the y-axis direction */
        colour += tex2D(blurredSampler, IN_TEXTURE + (YOffset * pixel_number)) *
            GaussianWeights[pixel_number];

        colour += tex2D(blurredSampler, IN_TEXTURE - (YOffset * pixel_number)) *
            GaussianWeights[pixel_number];
    }
    return colour; }
```


The final pixel shader combines the blurred and original textures, applying a tone mapping operation to the result. It starts by performing two texture colour lookups using declared samplers for both, one for the original, `originalSampler`, and another for the blurred image, `blurredSampler`. It then performs a linear interpolation, computing the average of the two colour samples. Following this, we calculate the distance of the current pixel to the centre of the screen. As discovered through experimentation (by tweaking the values until “it looked right”), this value to the power of 3.8 is then multiplied by the linearly interpolated colour and an exposure value, the subsequent result to the power of 0.5 being our final HDR pixel colour:

```
float4 ToneMappingPixelShader(float2 IN_TEXTURE: TEXCOORD0) : COLOR0
{
    float4 nonBlurredTexture = tex2D(originalSampler, IN_TEXTURE);
    float4 gaussianTexture = tex2D(blurredSampler, IN_TEXTURE);
    float4 colour = lerp(nonBlurredTexture, gaussianTexture, 0.5f);
    float pixelDistance = 1 - dot(IN_TEXTURE - 0.5f, IN_TEXTURE - 0.5f);
    colour = pow(colour * pow(pixelDistance, 3.8) * exposure, 0.5);
    return colour;
}
```

Using an exposure ranging from “0.0” to “1.5” will generally result in an under exposed image while an exposure of “2.0” to “4.0” will result in a properly exposed image. Increasing the exposure even more will lead to an overexposed image.

3.6 Shadows

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. 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 3.14 illustrates shadow generation, specifically the implementation of stencil shadow volumes – a popular shadow rendering technique.

Please note, the MSc dissertation, *An Empirically Derived System for High-Speed Shadow Rendering* (Rautenbach, 2008), offers a detailed look at shadow rendering and

that much of the information in this section has been sourced from it. Also, Appendix D presents the theory behind numerous real-time shadow rendering algorithms and techniques with the particular focus being on the rendering of shadows by means of stencil shadow volumes and depth stencil testing. The sections below present the implementation details of various shadow rendering algorithms (as used in the presented rendering engine), 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 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.

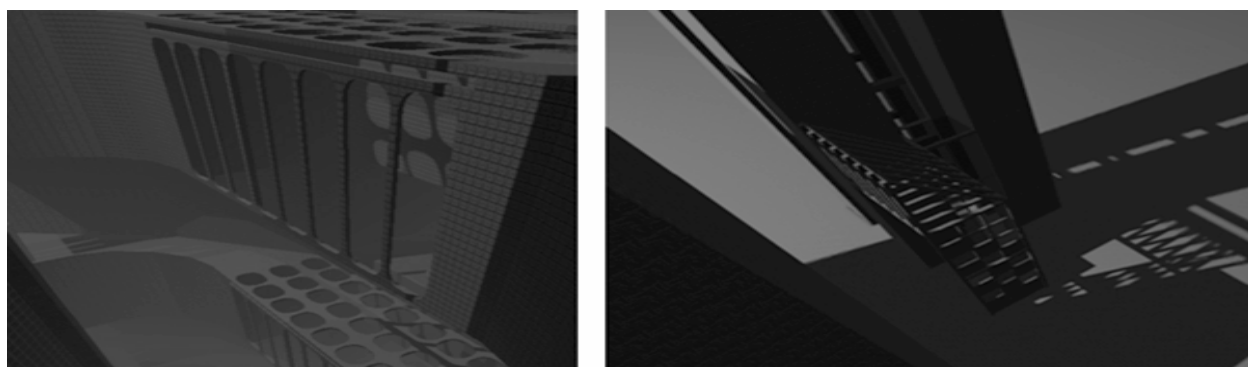


Fig 3.14 Example of stencil shadowing – note the overlapping shadows in the first image.

3.6.1 Stencil Shadow Volumes

Before looking at the stencil shadow volume implementation, it is necessary to discuss the stencil buffer and the depth-stencil testing process; two concepts crucial for the implementation of stencil shadow volumes. Figure 3.15 shows a shadow rendered by means of stencil shadow volumes.



Figure 3.15 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 Stencil Buffer

The stencil buffer is a buffer located on the 3D accelerator/video card 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) (Bell, 2003).

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 (Rossignac and Requicha, 1986).

Enabling Depth-Stencil Testing

Before initialising the stencil buffer it is important to set the depth stencil format 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 (Direct3D Utility Framework) structure. DXUT is a high-level framework built on top of Direct3D and it provides a series of functions, callbacks, structures, constants and enumerations that 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 depth-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_STENCIL_OP_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_STENCIL_OP_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_STENCIL_OP_DESC FrontFace;
    D3D10_DEPTH_STENCIL_OP_DESC BackFace;
} D3D10_DEPTH_STENCIL_DESC;

typedef struct D3D10_DEPTH_STENCIL_OP_DESC {
    D3D10_STENCIL_OP StencilFailOp;
    D3D10_STENCIL_OP StencilDepthFailOp;
    D3D10_STENCIL_OP StencilPassOp;
    D3D10_COMPARISON_FUNC StencilFunc;
} D3D10_DEPTH_STENCIL_OP_DESC;
```

The default values, including the alternatives, for the members of the **D3D10_DEPTH_STENCIL_DESC** structure are given in the table below (Microsoft, 2011):

Depth-stencil state	
DepthEnable	<i>TRUE (default)</i>
	<i>FALSE (alternative)</i>
DepthWriteMask	<i>D3D10_DEPTH_WRITE_MASK_ALL (default)</i> (enables writing to the depth-stencil buffer)
	<i>D3D10_DEPTH_WRITE_MASK_ZERO (alternative)</i> (disables writing to the depth-stencil buffer)
DepthFunc	<i>D3D10_COMPARISON_LESS (default)</i> (the test passes if the new data < existing data)
	<i>D3D10_COMPARISON_NEVER (alternative)</i> (no depth test is performed)
	<i>D3D10_COMPARISON_EQUAL (alternative)</i>

	(the depth test passes if the new data == existing data)
	D3D10_COMPARISON_LESS_EQUAL (alternative)
	(the depth test passes if new data <= existing data)
	D3D10_COMPARISON_GREATER (alternative)
	(the depth test passes if new data > existing data)
	D3D10_COMPARISON_NOT_EQUAL (alternative)
	(the depth test passes if new data != existing data)
	D3D10_COMPARISON_GREATER_EQUAL (alternative)
	(the depth test passes if new data >= existing data)
	D3D10_COMPARISON_ALWAYS (alternative)
	(the depth test is always performed and always passes)
StencilEnable	FALSE (default)
	TRUE (alternative)
StencilReadMask	D3D10_DEFAULT_STENCIL_READ_MASK (default)
StencilWriteMask	D3D10_DEFAULT_STENCIL_WRITE_MASK

Table 3.1 Default and alternative depth-stencil states.

Table 3.2 lists the `D3D10_DEPTH_STENCIL_OP_DESC` structure's possible stencil operations. These operations can be specified depending on the outcome of the stencil test. The `D3D10_DEPTH_STENCIL_OP_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_STENCIL_OP_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;
```

Next the depth stencil state (encapsulating all the above defined information for the pipeline stage determining the visible pixels) is set. This is done via 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 Direct3D 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 resource. Refer to the DirectX SDK documentation (Microsoft, 2011) 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 this case) and the third is the address of a pointer to the specified texture (`pDepthStencilBuffer`):

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

The final 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 against which the depth-stencil test is to be done:

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

Next the view mechanism is used to describe how the Direct3D depth-stencil resource will be handled (viewed) by the pipeline. In this case we Direct3D's "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-integer or signed-integer to floating-point format. The given source code implementation uses the **DXGI_FORMAT_D32_FLOAT** format (a 32-bit floating-point format). The second member, **ViewDimension**, is used to determine the depth-stencil resource 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).

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 `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);
```

The 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 3.16 and Figure 3.17, respectively.

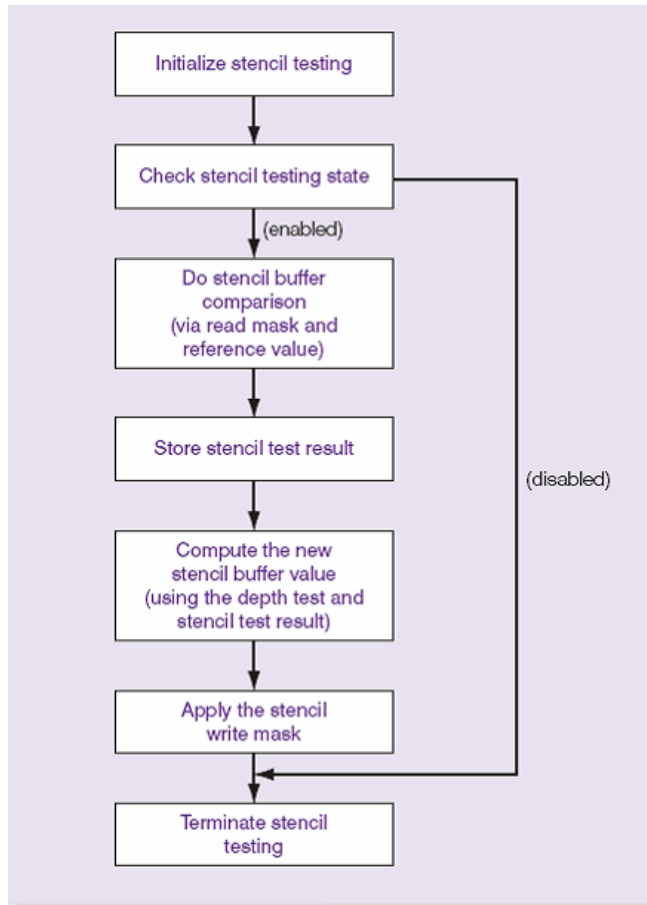


Figure 3.16 The stencil testing process,

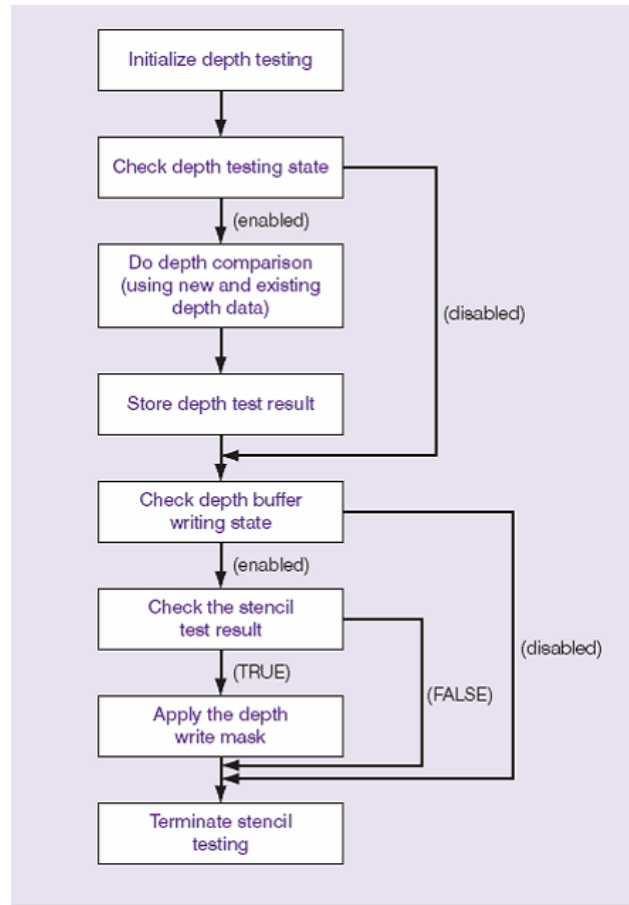
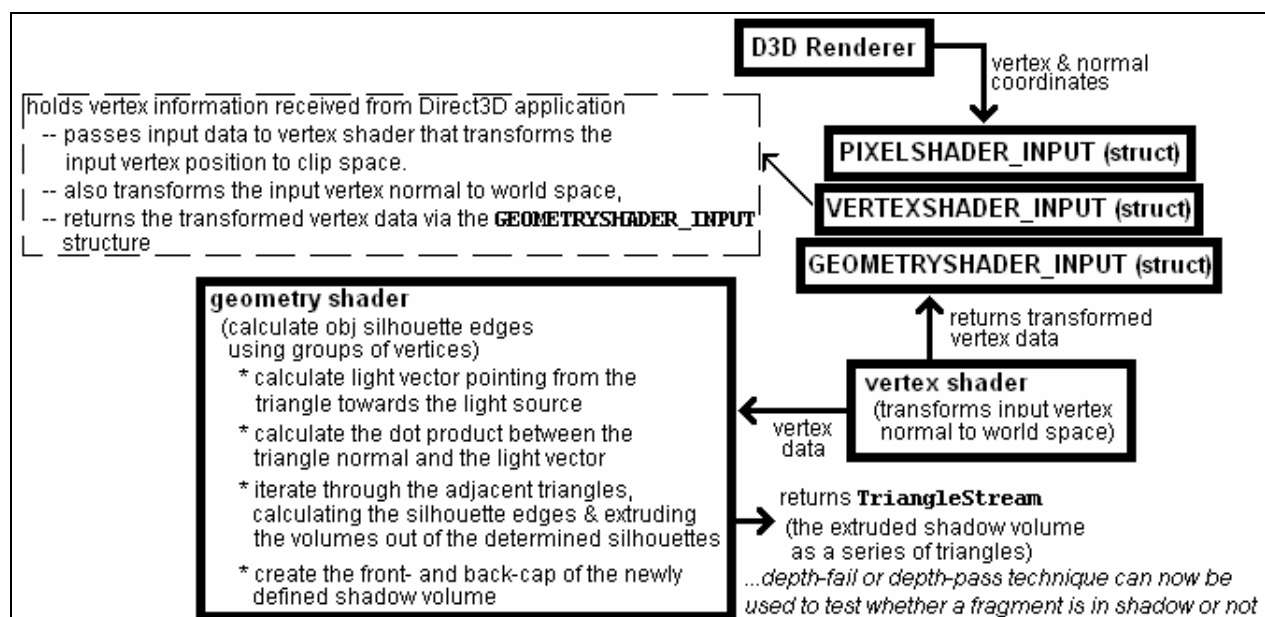


Figure 3.17 The depth testing process.

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. A shader is used to calculate the silhouette edges of an object with respect to a light source. 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.

The stencil shadow volume shader program 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 a geometry shader is written to determine an object's silhouette edges using groups of vertices, each group consisting of two shared vertices and one neighbouring or adjacent vertex. This shader function also receives an un-normalised triangle normal (**normal**) as input. It returns 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, just a note on the **triangleadj** input primitive type. This newly supported (DirectX 10 and later) geometry shader type flags every other vertex as an adjacent vertex (a **triangleadj** primitive is defined by six vertices, with the adjacent vertices being indexed as 1, 3, 5, for example), 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 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[i],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();

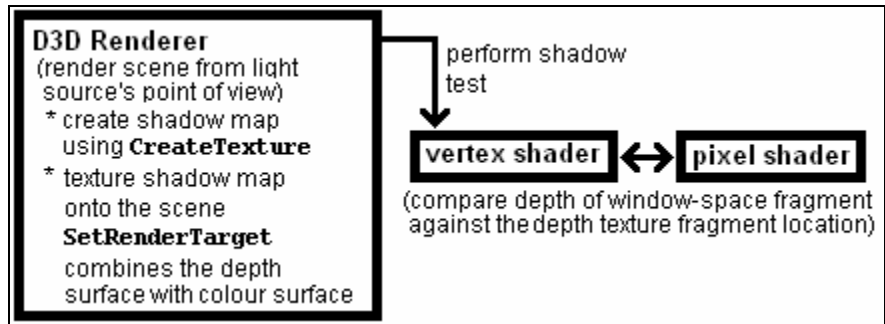
//now calculate the furthest cap
for(int k = 4; k >= 0; k -= 2)
{
    float3 farCapPosition = vertex[k].Loc + shadowExtrusionAmount *
                                normalize(vertex[k].Loc - LightPosition);

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

```

The previously discussed depth-fail or depth-pass technique can now be used to test whether a fragment is in shadow or not. The chosen depth-stencil test can be implemented using native Direct3D 10 structures and functions as listed in the previous section. 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.

3.6.2 Implementing Shadow Mapping



This section considers the presented engine’s shadow mapping algorithm. Figure 3.18 shows a high-resolution shadow map.



Figure 3.18 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.

Shadow mapping, unlike shadow volumes, does not 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 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 well: "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)
{
    colour = Depth.x / Depth.y; // the depth is actually x/y}
}
```

3.6.3 Hybrid and Derived Approaches

We now present a high-level overview of a number of hybrid stencil shadow volume/shadow mapping approaches (no code walkthroughs are given as these algorithms are basic combinations of the previously discussed stencil shadow volume and shadow mapping techniques). Please see the accompanying CD for source code implementations.

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. McCool 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 is not 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).

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.

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 performing the following steps:
 - a. Find the ridge edges.
 - b. Connect the ridge edges to form loops.
 - c. Determine the angular coordinates (r, θ, ϕ) of all vertices positioned on ridge edge loops.
 - d. Identify the vertices with local peaks in θ . Ridge edge loops are sliced along θ with local peaks being specific points in the loop.
 - e. Append all the points of edges to the lookup table until a minimum in θ is reached (by starting from the identified peaks).
 - f. Insert the hidden edges in the lookup table.
 - g. Perform 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 is in shadow or not. This is done for each scan line – see Figure 3.19.
- 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, theta and phi
If table entry at theta exists
    nextX = END
    For all pairs (phi_i, phi_j) of table entry
        If (phi_i <= phi <= phi_j AND r_i <= r)
            nextX = xEquivalentOf(phi_j)
            return TRUE
        Else if(phi < phi_i)
            tempX = xEquivalentOf(phi_i)
            If (tempX < nextX)
                nextX = tempX
    return FALSE
Else
    nextX = END
    return FALSE

```

Figure 3.19 The query function as given by Thakur et al (2003).

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, 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 (Fuchs et al, 1980). 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.

3.7 Physics

Video games originally featured a very small amount of physics simulation. A game such as Breakout (released by Atari in 1976 and shown in Figure 3.20) illustrates the point. It incorporated a limited degree of collision detection and response to simulate the

destruction of bricks upon collision with a ball, as well as the bouncing of this ball upon impact with the movable paddle.



Figure 3.20 A Breakout clone (source code available on the accompanying CD).

During the 1990s, concepts such as gravity and other fundamental laws of physics started steadily finding their way into games (Hecker, 2000). It was not, however, until the release of games like *Valve Software's Half Life 2* that true physics simulation significantly contributed to the overall game play experience. *Half Life 2* included numerous physics-based puzzles where the player, for example, had to use gravity by removing bricks from one end of a pulley system to lower the other end, etc. Physics has thus found its way into games for the realistic simulation of object-player interaction as well as for the animation of objects based on exerted forces and environmental resistance.

One interesting development in the world of physics has been the emergence of dedicated Physics Processing Units or PPU's. These dedicated physics microcontrollers act in much the same way as GPUs, in this case relieving the CPU of all physics and math calculations. AGEIA (acquired by NVIDIA) did a lot of work on Physics Processing Units and invented the *PhysX* (shown in Figure 3.21) – a PPU that accelerates physics calculations by offloading them from the CPU to PPU. This PPU is limited to acceleration of AGEIA's own physics engine – the *PhysX SDK* (a real-time physics engine middleware SDK now known as *Nvidia Physix* and available on CUDA-enabled GeForce GPUs).

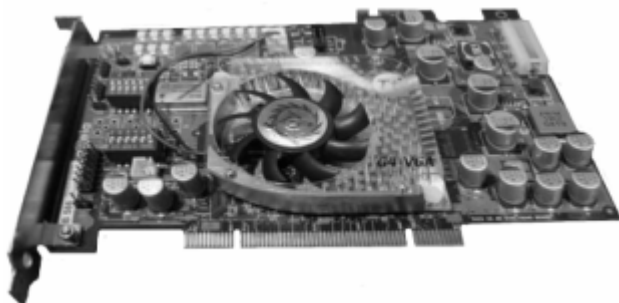


Figure 3.21 Asus-based AGEIA's PhysX PPU card.

Prior to acquiring AGEIA, NVIDIA competed against the PhysX PPU by accelerating the Havok FX SDK (a specialised version of the Havok physics engine used in Half Life 2) to utilise the GPUs in ATI and NVIDIA video cards for physics simulations.

3.7.1 The Role of Newton's Laws

Most physics simulations are based on *Newton's laws of motion* – three laws describing the relationship between the forces influencing a rigid body and the resulting motion of this body. The performance of a physics simulation is heavily dependant on the number of bodies being simulated since the exact modelling of these laws requires so much processing power that even the most powerful computers can eventually grind to a halt as the number of bodies increase. Newton's laws of motion can be summarised as follows:

1. The first law: law of inertia
 - A body will remain in its state of rest or uniform motion in a straight line, unless an external force causes a change to that state.
2. The second law: law of acceleration
 - The net force of a particle is the rate of change of its linear momentum.
 - Momentum is the mass of the body multiplied by its velocity.
 - The force on a body is thus its mass multiplied by its acceleration ($F=m.a$).
3. The third law: law of reciprocal actions
 - To every action there is an equal and opposite reaction.

Computer games will rarely implement physics or Newton's laws of motion down to the letter. Doing so will leave little if any processing power for the game's AI, networking, game loop, etc. as slowdowns often occur when these laws are applied to a large number of objects in a scene. We will thus rather outline the physics needed and simulate the required effects as close to real life as possible, hence creating an extremely close approximation but using a lot of optimisations and assumptions to simplify the original laws of motion. The presented rendering environment features realistic object interaction based on Newton's Laws (all objects react based on forces exerted and environmental resistance) as well as a particle system inheriting from the physics system.

3.7.2 Particle Effects

The presented rendering engine's particle system is a graphics subsystem used to simulate certain natural phenomena such as fire, smoke, sparks, explosions, dust, trail effects (Figure 3.22), etc.

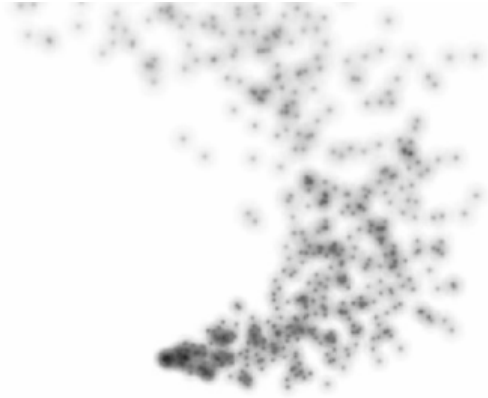


Figure 3.22 Rendering trails with a particle system.

The particle system is implemented using three stages, namely, the setup stage, the simulation stage and the rendering stage (Crossno and Angel, 1997). The *setup stage* involves specification of the particle system's spatial position and area of constraint – parameters controlled by the *emitter*. The emitter also controls the particle creation rate, that is, the rate at which new particles are injected into the system. Each particle has a specific time to live, after which it is destroyed. The *simulation stage* takes care of particle rendering rates, particle spawning position (mostly randomised between some minimum and maximum coordinate range), particle properties (such as particle colour, velocity, etc) and positioning of the emitter. This stage also keeps track of each particle to check whether a specific particle has exceeded its lifetime. Each particle has an initial velocity and is translated based on some sort of physics model or simply by adding velocity to its current spatial position. Collision detection, in general, is also possible at this stage but rarely implemented (Hubbard, 1996). Following the simulation state, each particle is rendered as either a coloured point, polygon or as a mesh. Figure 3.23 shows the generation of particles over time.

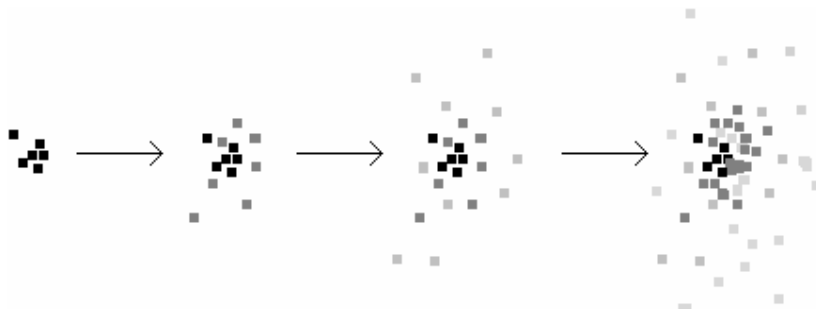


Figure 3.23 Particles being generated over time.

The presented particle system, based on the rules of physics, uses the following standard equations to calculate each particle's velocity and position:

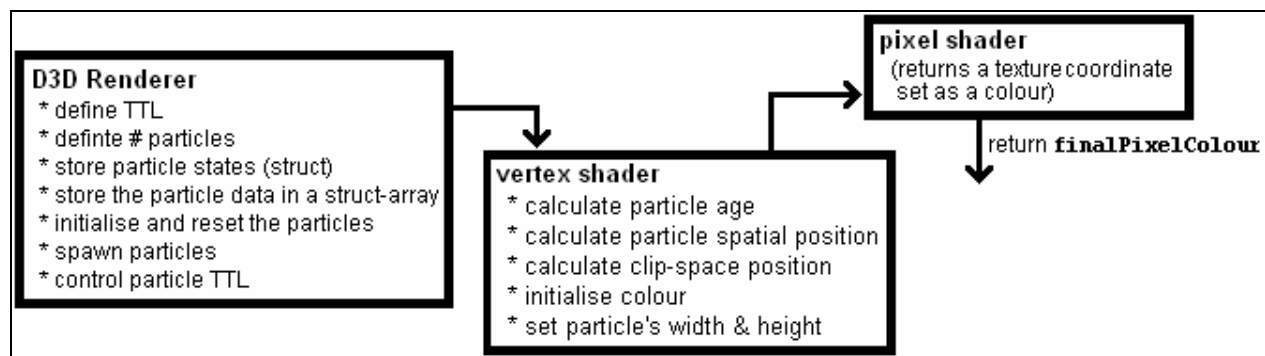
$$V_{new} = V_{old} + g \times t$$

$$Pos_{new} = Pos_{old} + (v_{old} \times t) + \left(\frac{1}{2} \times a \times t^2\right),$$

The above given equations factor in the initial motion of the particle and its trajectory and the overall effect of gravity where Pos_{new} is the particle's final position, Pos_{old} its initial position, V_{new} its final velocity, V_{old} its initial velocity, a the particle's acceleration and t the change in time. Using these equations we start by initialising each particle's initial position and velocity. These values will be assigned to a particle when it is generated by the emitter.

Implementing a particle system in C++ is quite a tedious task due to the necessitated creation of a data structure for the storage of particle data (particle state, spawning coordinates and velocity, current velocity and position, rendering colour, etc). We also need member functions for the setup, initialisation, generation, rendering as well as the cleanup of particles the moment their time to live expires (Gallagar, 1995). Using shaders on the other hand allow us to easily create a particle system – as illustrated by the vertex and fragment shader-based particle system code given in the following section.

3.7.3 Particle System Implementation



We now present the shader implementation of the proposed engine's particle system.

The first step is to specify the name of the vertex program's entry function, `particle_vertex`. It has the following signature (where `PSIZ` is just a binding semantic for point size):

```
void particle_vertex(float4 initialParticleVelocity : TEXCOORD0,
```



```
float4 particleAcceleration : TEXCOORD1,  
float4 initialParticlePosition : POSITION,  
float particleCreationTime : TEXCOORD2,  
  
out float outputParticleSize : PSIZ,  
out float4 outputParticleColour : COLOR,  
out float4 outputParticlePosition : POSITION,  
  
/* parameters supplied by the application program */  
uniform float4x4 totalRunningTime,  
uniform float4x4 modelToWorldTransformation,  
uniform float4x4 modelviewProjection)
```

We start by calculating the particle's age. (The particle's time of creation is subtracted from the total time the simulation has been running – as sent from the application to the shader):

```
/* calculate the amount of time the particle has been active */  
float particleTime = totalRunningTime - particleCreationTime;
```

The particle's spatial position is calculated using the standard physics equation given above:

```
float4 finalParticlePosition = initialParticlePosition +  
    initialParticleVelocity*particleTime +  
    (0.5f)*particleAcceleration* pow(particleTime, 2);
```

Next the clip-space position is calculated:

```
/* transform the vertex position into homogeneous clip- space coordinates */  
outputParticlePosition = mul(modelviewProjection, finalParticlePosition);
```

All that remains now, before the particle's width and height are set, is to initialise its colour:

```
/* set the particle colour to green */  
outputParticleColour = (0, 0.5, 0, 1);
```

The final operation is to set the particle's width and height:

```
/* set the particle's size */  
float3 outputParticleSize = 0.5;
```


A fragment shader function, `particle_fragment`, that simply returns a texture coordinate set as a colour, is now defined:

```
void particle_fragment(float4 inputParticleColour : TEXCOORD0,out float4 colour: COLOR)
{
    /* set the colour */
    return colour;
}
```

The above implementation can be used as the core of a particle generator (to generate particles as shown in Figure 3.23). Regarding the particle system's C++ implementation, we have to initialise the number of particles, create a list of particle start times, spawn particles in a semi-random fashion (within the area of a spawning point) and destroy particles whenever their time limit is exceeded:

```
/* start by limiting the number of particles at any given time to 600 */
#define TOTAL_NUMBER_PARTICLES 600

#define TTL 30; /* set the maximum time to live */

/* create a structure to store the particle states */
typedef struct Particle
{
    float initialParticlePosition_[3];
    float initialParticleVelocity_[3];
    float particleAcceleration_;
    float particleTime_;
    bool isAlive;
} Particle;

/* store the particle data in a struct-array */
Particle particleStartData[TOTAL_NUMBER_PARTICLES];

/* return a random double within the passed range */
double GetRandomDouble(double low, double high)
{
    return ((double)rand()/(RAND_MAX+1.0))*(high - low) + low;
}

/* function to initialise and reset the particles */
void InitParticleSystem()
{
    /* initialise each particle */
```

```

for(int i = 0; i < TOTAL_NUMBER_PARTICLES; i++)
{
    /* set the initial starting position (x, y, z) */
    particleStartData[i].initialParticlePosition_[0] = 0.0;
    particleStartData[i].initialParticlePosition_[1] = 0.0;
    particleStartData[i].initialParticlePosition_[2] = 0.0;

    /* set the initial velocity (x, y, z) */
    particleStartData[i].initialParticleVelocity_[0] = 0.0;
    particleStartData[i].initialParticleVelocity_[1] = 0.0
    particleStartData[i].initialParticleVelocity_[2] = 0.0;

    /* set the gravity acceleration */
    particleStartData[i].particleAcceleration_ = -9.8;

    /* start the particles at a random time */
    particleStartData[i].particleTime_ = GetRandomDouble(0, 5);

    /* activate particles */
    particleStartData[i].isAlive = false;
}
}

/* function to spawn particles */
void spawnParticles()
{
    /* spawn particles */
    for(int j = 0; j < TOTAL_NUMBER_PARTICLES; j++)
    {
        if((particleStartData[j].isAlive == false) &&
            (particleStartData[j].particleTime_ < TTL))
        {
            /* change the particle velocity (x, y, z) */
            particleStartData[j].initialParticleVelocity_[0] = GetRandomDouble(-1,1);

            particleStartData[j].initialParticleVelocity_[1] = GetRandomDouble(-0.5,
                                                                              0.5);

            particleStartData[j].initialParticleVelocity_[2] = GetRandomDouble(0,
                                                                              2.5);

            particleStartData[j].isAlive = true; // flag the particle as active
        }
    }
}

```

```
}

/* function to decrease a particle's time to live */
void decreaseParticleTTL()
{
    /* destroy particles */
    for(int k = 0; k < TOTAL_NUMBER_PARTICLES; k++)
    {
        if((particleStartData[k].isAlive == true) &&
            (particleStartData[k].particleTime_ < TTL))
        {
            particleStartData[k].isAlive = false; // flag the particle as inactive
            particleStartData[k].particleTime_ += 0.01; //increase the particle's ttl
        }
    }
}
```

The above given functions can now be combined with the featured vertex and fragment shader to render live particles as shown in Figure 3.23.

3.8 Post-Processing

The presented rendering engine uses post-processing or quality-improvement image processing (through the use of pixel shaders) to add additional effects such as bloom lighting, motion blur, ambient occlusion, depth of field and halo effects. Post-processing quality scaling is discussed in Part II of this thesis.

3.9 Summary

The chapter presented our modular rendering engine as a scalable interactive testing environment and solution for the rendering of computationally intensive 3D environments. It extended chapter 2's basic DirectX 10 3D interactive environment through the addition of several subsystems, specifically: HLSL shaders, local illumination, reflection and refraction, HDR lighting, additional shadow rendering algorithms, physics simulation, particle effects and post-processing special effects.

Part II of the thesis categorises these presented approaches and rendering groupings based on the level-of-detail/rendering quality and the associated computational impact. It also focuses on the critical analysis and detailed benchmarking of the presented rendering and simulation techniques – the data to be used by our fuzzy-based selection and allocation system.



Part II

Maximising the Quality and Performance of A Real-time Interactive Rendering System

Benchmarking the Rendering Algorithms and Techniques

Chapter 4 presents the critical analysis and benchmarking of the previously discussed rendering algorithms and techniques as utilised by our interactive rendering engine. The empirical analysis presented in this chapter allows us to explore in the next chapter the practicality and performance benefits of a dynamically scalable interactive rendering engine in which GPU-CPU utilisation, as a secondary proof of concept approach, has been unified.

Outline:

- Benchmarking mechanism
- Evaluation criteria
- Algorithm comparison

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 rendering subsystems. This benchmarking system basically functions as a plug-in to the previously discussed rendering engine where real-time performance data are streamed to a file-based database for post-processing and analysis. Figure 4.1 gives a visual representation of this system.

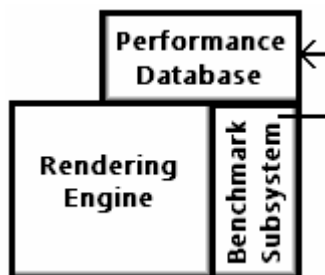


Figure 4.1 The rendering engine, benchmarking system and performance database.

Critical analysis was performed via scripted camera movement, object 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 Rendering Subsystem Evaluation Criteria

The set of criteria used to evaluate the presented rendering techniques (such as cube mapping, post-processing effects and stencil shadow volumes) is now presented. The given evaluation criteria were selected with the aim of assessing the relationship between rendering quality and performance. This assessment provides the basis of the system presented in Chapter 5 in which the dynamic selection of algorithms as well as CPU-GPU process allocation (for cube mapping and physics processing) is used to control performance and quality. 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 <i>(performance)</i>	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, for example, an increase in the number of light sources and the shadow casting model's polygonal complexity).
Rendering Accuracy and Detail <i>(quality)</i>	Determining whether, for example, a shadow is cropped and/or skewed properly, and accurately projected onto other models and surfaces, or whether a reflection is accurate allows for the evaluation of rendering quality.
CPU/GPU Utilisation <i>(performance/quality)</i>	Comparing, where applicable, a standard GPU-driven implementation to the same implementation being run on a CPU (with utilisation of modern multi-core architecture); allows for the evaluation of maximized parallelism versus conventionally GPU-based rendering.

Table 4.1 Evaluation criteria

4.3 Algorithm Comparison

This section compares the presented engine's core rendering elements. It also lists the observed results with specific emphasis on the most appropriate application areas.

To gather the necessary results, all algorithms were implemented for a number of scenes. In each case, unless otherwise stated, 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 (Figure 4.2). The test system had the following configuration:

NVIDIA GeForce™ GTX 570 (1280MB GDDR5) (Video Card),
Intel Core i5 650 @ 3.2GHz+ (Dual Core Processor),
4.0GB (Memory),
1920x1080 (Screen Resolution).



Figure 4.2 The 599-face 'tiger' mesh, 1628-face 'car' mesh, 4136-face 'shapes' mesh and a 9664-face 'battleship' mesh.

4.3.1 Shadows

The presented evaluation focuses on a number shadow rendering algorithms (discussed in section 3.6), 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 and spatial subdivision approach. Please note that the mean performance of each algorithm is shown (performance data has been average over the four models due to individual behaviour showing identical patterns). Also, for the detailed critical analysis, please see the MSc dissertation, An Empirically Derived System for High-Speed Shadow Rendering (2008).

Starting out, it is important to note that Rautenbach et al’s 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. This octree-based algorithm will thus outperform all other algorithms where light sources are not added, moved or removed.

In Figure 4.3 the frame rates achieved via the implementation of spatial subdivision is compared to that obtained using the Heidmann algorithm. It is clear from the data that Rautenbach et al’s approach results in significantly better performance than the original stencil shadow volume algorithm (40% better for one light source and 200% better for eight).

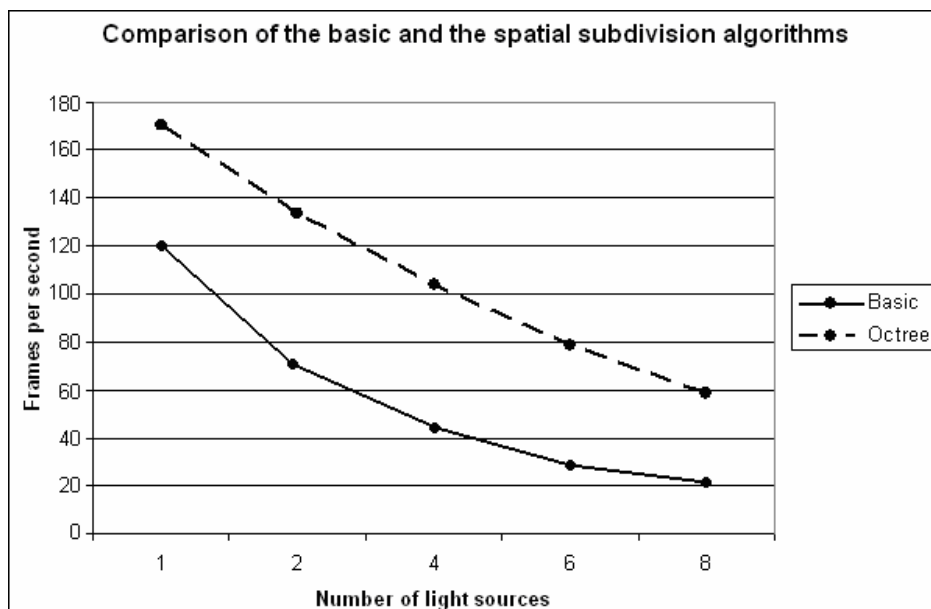


Figure 4.3 Comparison of Rautenbach et al’s spatial subdivision approach with the depth-fail stencil shadow volume approach.

In Figure 4.4, 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. Comparing a standard C/Direct3D implementation to the utilisation of Intel’s SSE2 instruction set allows for the evaluation of maximised parallelism (as offered by these instruction sets) versus conventionally sequentially executed routines. Intel’s SSE 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 SSE instruction set (specifically SSE2 found on the Pentium 4+ architecture) also adds 64-bit floating point and 8/16/32-bit integer support.

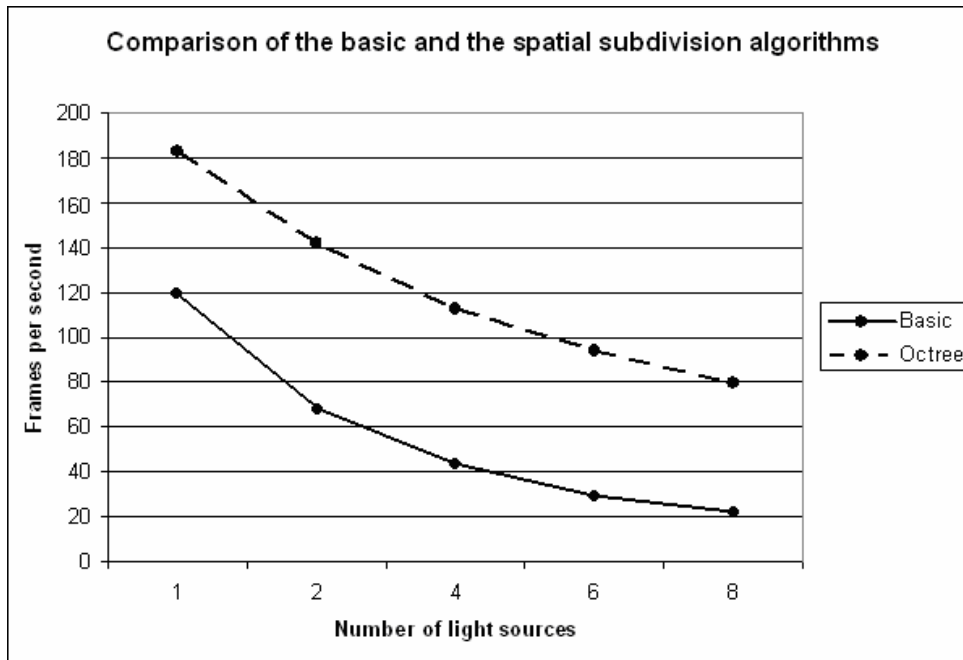


Figure 4.4 Comparison of Rautenbach et al’s extended approach with the depth-fail stencil shadow volume approach.

From the results given in Figure 4.4, it is clear that Rautenbach et al’s spatial subdivision approach combined with SSE 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. The presented rendering engine will use the spatial subdivision approach coupled with SSE2 utilisation for all environmental areas lit using static light sources.

From Figure 4.5, listing the mean performance comparison of all the shadow algorithms listed in section 3.6, it is clear that 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. The presented rendering engine will use Chan and Durand's algorithm for all scenes consisting of eight or less dynamic light sources when high-quality shadows are required.

Furthermore, the shadow mapping algorithm is observed to perform only slightly worse than Chan and Durand's (2004) algorithm (when rendering scenes consisting of just one light source). That said, the critical analysis implementation does render low-resolution shadow maps. However, increasing this shadow map resolution will have a net-negative impact on the scene's overall rendering performance. Shadow mapping (with average shadow resolution) is used 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.

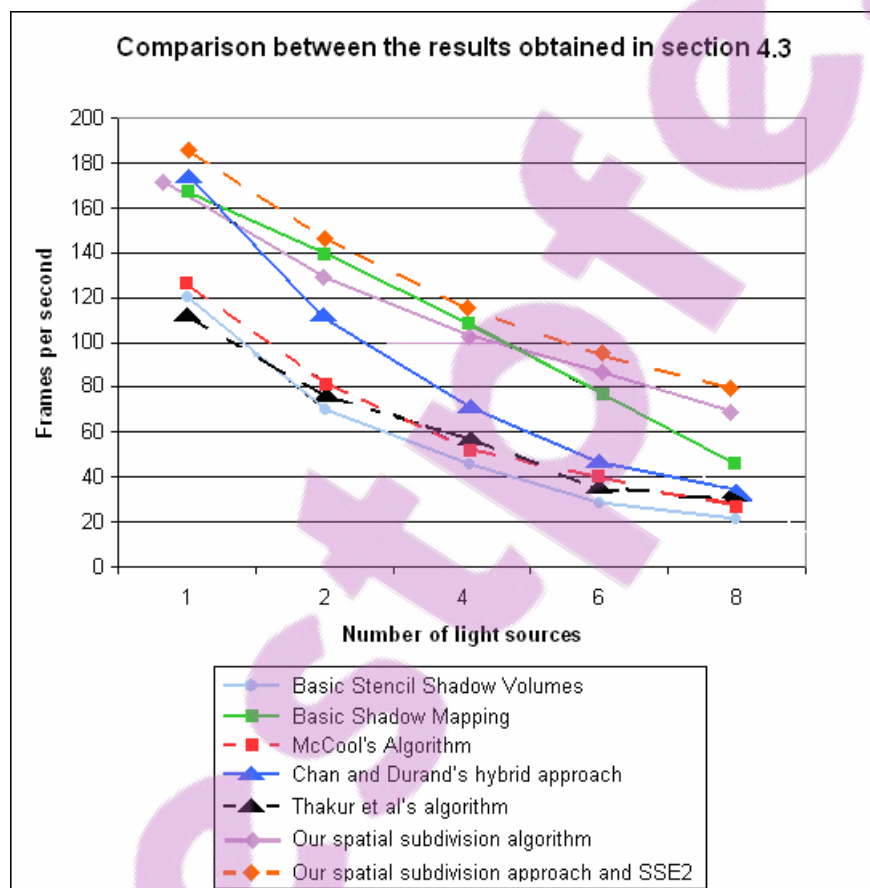


Figure 4.5 Comparison of all the shadow algorithms listed in Chapter 3 (1-8 light sources).

The previous comparison (given in Figure 4.5) 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 the frames per second performance. Figure 4.6 shows these results.

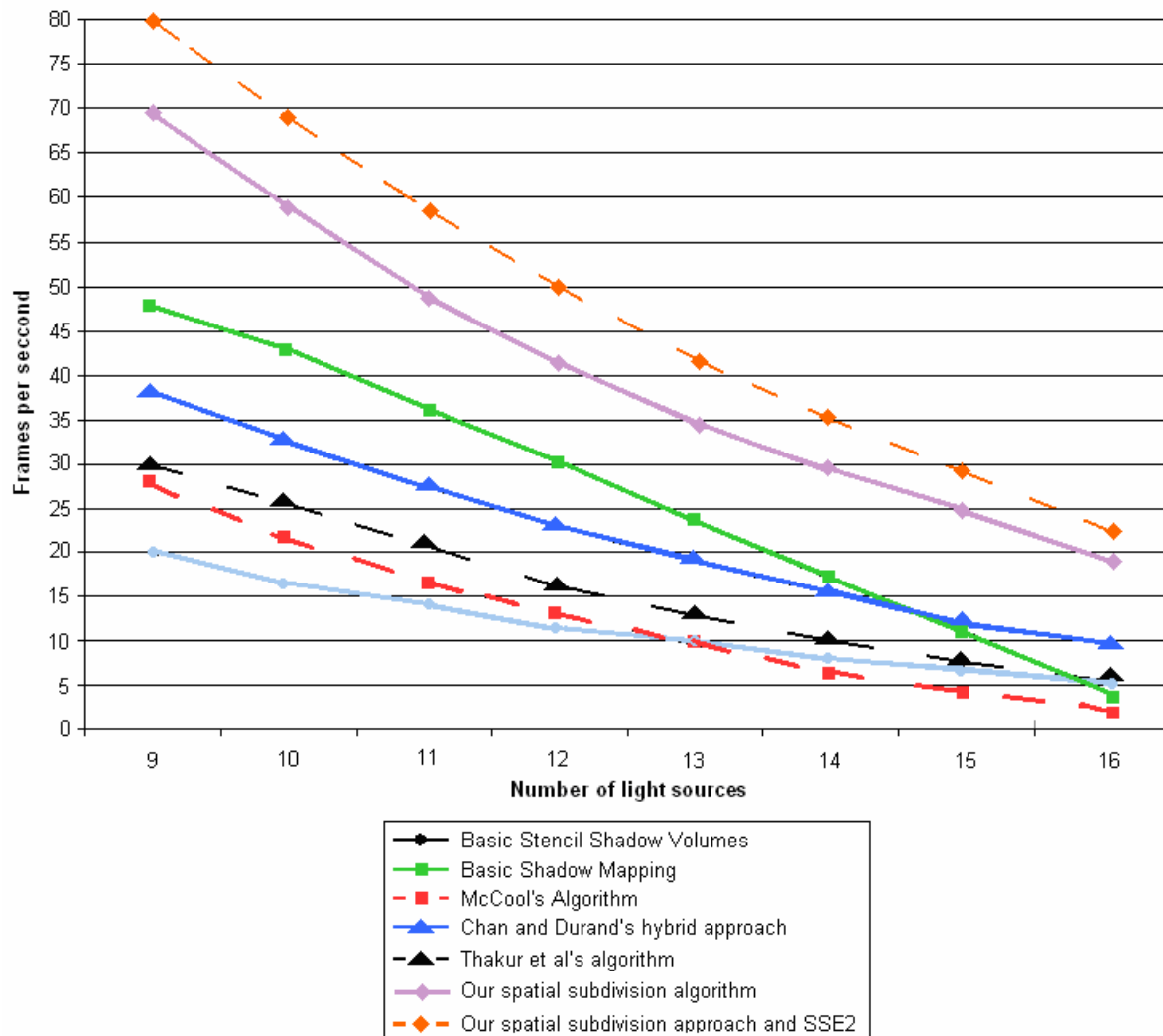


Figure 4.6 Comparison of all the previously listed algorithms (9-16 light sources) – please note; Rautenbach et al's spatial subdivision algorithm was analysed in a statically lit environment, thus resulting in its high performance (its performance is comparable to the basic stencil shadow volume algorithm in situations where dynamic lighting is implemented).

Considering Figure 4.6, Rautenbach et al's spatial subdivision approach coupled with the SSE2 instruction set is once again observed to outperform all the other algorithms. This algorithm is, as mentioned, only amenable to environments utilising static lighting –

making the comparison a bit bias. The presented rendering engine 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 is not as important (see Figure 4.5 and 4.6). The presented rendering engine will 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. Chan and Durrand's algorithm will also be used for all scenes consisting of nine or more dynamic light sources when high-quality shadows are required.

Shadow selection is based on the optimisation of the rendering frame rate and shadow quality. The presented rendering engine 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 4.2 summarises the algorithms of choice based on the algorithmic comparison and scene conditions such as view distance, dynamic/static light conditions and number of light sources.



Most Appropriate Algorithm	Conditions
Rautenbach et al's (2008) spatial subdivision approach coupled with SSE2 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. <i>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.</i>
McCool's (2000) and Thakur et	The second best choice when dealing with

al's (2003) algorithm.	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 4.2 Algorithms of choice based on the presented critical analysis.

4.3.2 Shaders

The presented shader evaluation focuses on a number of shader implementations and lighting approaches (please refer to Appendix B and C, respectively, for a background discussion on shaders and lighting as well as various reflection models). These implementations, listed in Table 4.3, are organised into four shader effect quality groups based on each algorithm or technique's standalone processor utilisation and visual effect quality. Algorithms and rendering approaches are grouped in order of increasing complexity. For example, basic directional lighting is a much simpler (and thus less computationally intensive) approach than a lighting model that adds ambient occlusion to a scene.

Grouping/Description	Rendered Scene Screenshot
<p>Low Shader Quality</p> <p>Consists of a simple light mapping shader implementation and a shader program enabling basic directional lighting (local illumination).</p>	
<p>Medium Shader Quality</p> <p>Extends the low shader quality grouping with the addition of a normal/bump mapping shader (as discussed in Appendix A, bump- or normal mapping is used for adding depth to pixels and thus creating a lighting-dependent bumpiness to a texture mapped), a shader used for the calculation and rendering of specular highlights (as discussed in Appendix C,</p>	

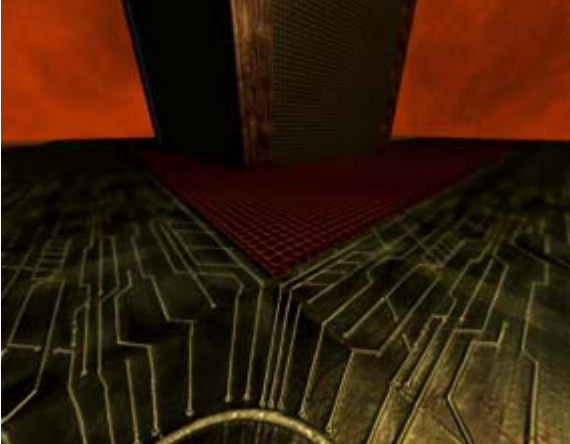

<p>specular reflection is characterized by bright highlights on the surface of an object reflected in the direction of the view vector) and a shader enabling volumetric fog.</p>	
<p>High Shader Quality</p> <p>Similar to the Medium Shader Quality group but replaces light mapping with a detailed lighting model shader that adds a shader for ambient occlusion (as a way to enhance the ambient light term such that shadows and light emission from local features are included).</p>	
<p>Very High Shader Quality</p> <p>Extends the High Shader Quality selection by replacing the previous lighting model with High Dynamic Range Lighting and parallax mapping (an enhancement of bump/normal mapping; “bumpy” textures will have more apparent depth and will thus appear more realistic).</p>	

Table 4.3 Shader effect quality groupings.

As discussed in section 3.5, high dynamic range lighting is the rendering of lighting using more than 256 colour shades for each of the primary colours. Thus, 16 to 32-bit colours per RGB channel are available for use (as opposed to the normal 8) – eliminating luminance and pixel intensity being clamped to a [0, 1] range. This allows the presented rendering engine the display of light sources over 100 000 times brighter than normally possible. HDR lighting results in the full visibility of both very dark and fully lit areas; unlike normal lighting, or *low dynamic range lighting*, where details are hidden in dark scenes when contrasted by a fully lit area. Using this form of lighting generally leads to a more vibrant looking scene. The inclusion of HDR Lighting (which, as shown, may greatly impact the performance of a scene) is controlled through the engine’s shader quality scaling (with quality relying on the GPU’s computational power).

Please see Figure 4.7 for the observed frames per second performance of each shader quality scaling group as the number of light source vary between 1 and 8. Figure 4.8 shows the same as the light sources increase from 9 to 16.

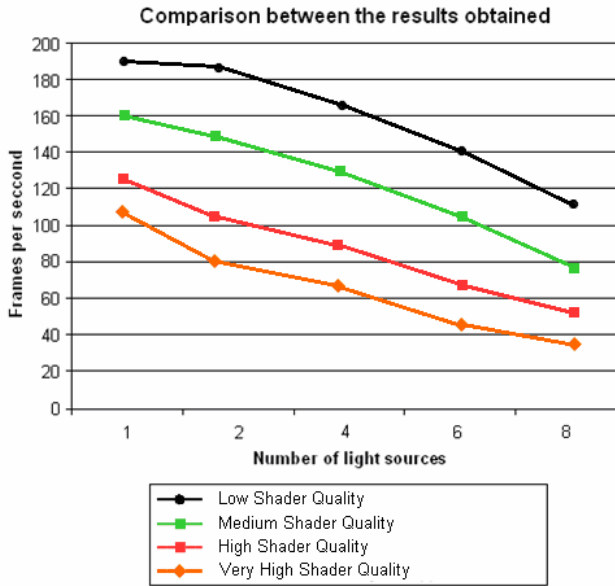


Figure 4.7 Comparison of all the previously listed quality scalings (1-8 light sources).

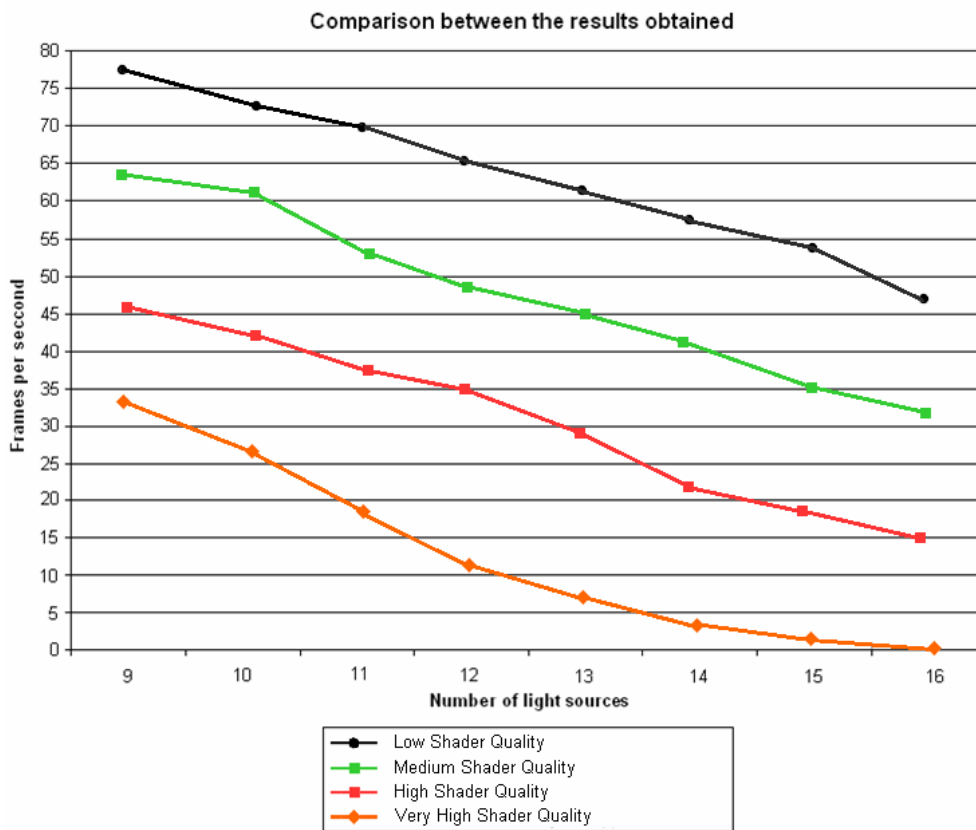


Figure 4.8 Comparison of all the previously listed quality scalings (9-16 light sources).

The first quality grouping is the best performing configuration and perfectly suited as a performance-orientated selection in situations where the GPU is being over-utilised or when faced with limited computational resources. As with the other shader groupings, this combination shows significant performance degradation when more light sources are added. It does, however, outperform all the remaining groups (the only cost being rendering quality).

The medium shader quality grouping performs only slightly worse than the first but given the visual quality benefits inherent to the utilisation of normal mapping, specular highlights and volumetric fog, it is clear that the first quality grouping should only be selected as a last resort effort to free up computational resources.

The third grouping performs relatively well when dealing with scenes featuring one to eight light sources and when high-quality special effects are required. The presented 3D engine will only utilise this shader grouping for scenes consisting of 14 light sources or less (as the FPS performance drops off significantly given further light source additions).

The final grouping gives similar performance figures and will only be utilised for scenes containing less than eight light sources and where the processing resources are available to facilitate HDR lighting and parallax mapping. That said, when there are few light sources, then, on the test system's hardware configuration, the FPS performance for even the very high shader quality grouping is well above the level at which the human eye can perceive any slowdowns. Running the same simulations with the light source count ranging from nine to sixteen shows a rapid decrease in the frames per second performance.

Shader selection is based on the optimisation of the rendering frame rate and rendering quality. The presented rendering engine will thus select the most appropriate shader grouping 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 scene being rendered. The rendering accuracy and detail of distant objects (for instance, distant normal mapping calculations) will carry less weight than those rendered relatively close to the viewer. Table 4.4 summarises the most appropriate shader quality selections based on our algorithmic comparison and scene conditions such as view distance, dynamic/static light conditions and number of light sources.

Most Appropriate Selection	Conditions
Low Shader Quality.	The GPU is heavily overburdened (significant slowdowns or FPS drops are observed) and additional computational resources are required by other core rendering elements.
Medium Shader Quality.	The GPU is fully utilised (a GPU running at 100%)

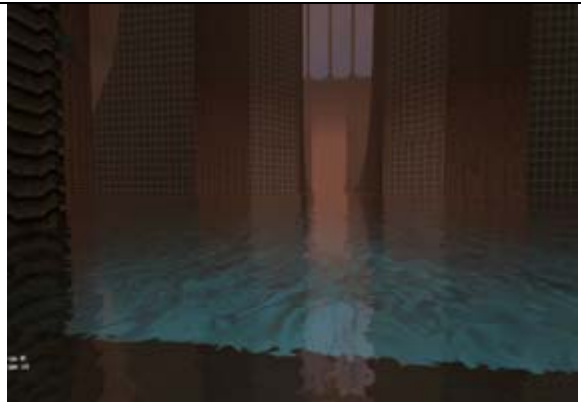
	but no additional computational resources are required (no slowdowns or noticeable FPS drops are observed).
High Shader Quality.	The GPU is not fully utilised and the scene consists of one to eight light sources and high-quality special effects are required but Very High Shader Quality would overburden the GPU.
Very High Shader Quality.	The scene contains less than eight light sources and the computational resources are available to facilitate true HDR lighting and parallax mapping.

Table 4.4 Shader quality selections based on the presented critical analysis.

4.3.3 Local Illumination

As discussed in the previous chapter, the presented rendering engine, in its most basic form, allows for the use of local illumination which, unlike global illumination, only considers the interaction between a light source and object. Local illumination is implemented using the diffuse reflection model, resulting in a uniformly lit scene. An HLSL pixel shader is implemented to calculate the lighting effect on each pixel in our scene.

The evaluation focuses on two basic implementations (divided into two performance-impacting groups, Low and High). Table 4.5 lists these two lighting scaling approaches with Figures 4.9 and 4.10 giving the observed performance of each.

Grouping/Description	Rendered Scene Screenshot
<p>Low Local Illumination</p> <p>Limits the number of light sources in an attempt to reduce GPU utilisation.</p>	

High Local Illumination

Lifts the lighting limitation imposed by the low lighting group and occludes local light sources (a technique used to approximate the effect of environment lighting as an attempt to simulate the way light radiates in real life).

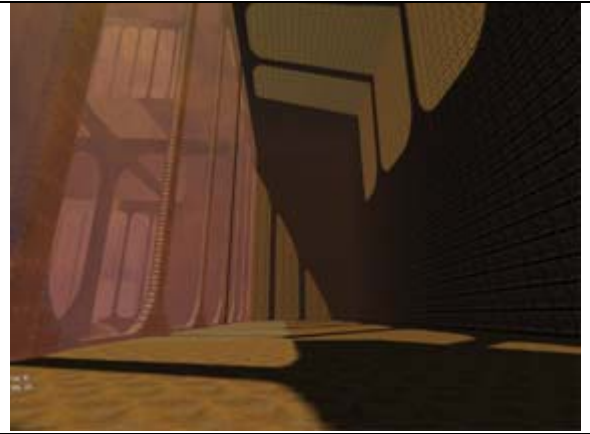


Table 4.5 Lighting quality groupings.

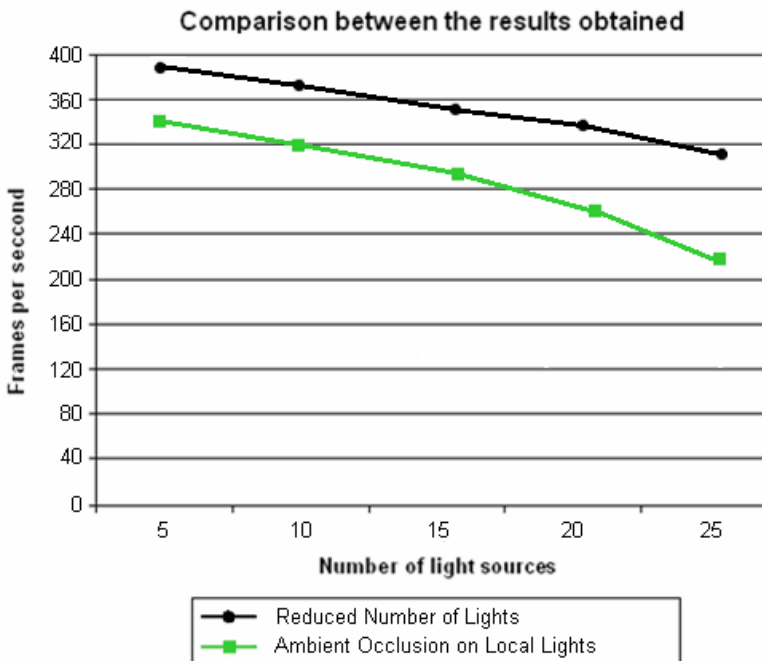


Figure 4.9 Comparison of all the previously listed quality scalings (5-25 light sources).

The first quality grouping, limiting the number of light sources, is perfectly suited as a performance-orientated selection in situations where the GPU is being over-utilised or when faced with limited computational resources (especially when rendering scenes consisting of 25 light sources or more). As with the other quality grouping, basic local illumination shows significant performance degradation as more light sources are added. The cost is not so much the scene’s overall rendering quality as it is a limit on the scene’s overall atmosphere and ambience (as can be observed by comparing the screenshots given in Table 4.5).

The high lighting quality grouping performs only slightly worse than the first but given the quality benefits inherent to the utilisation of ambient occlusion (as a way to enhance the

ambient light term such that shadows and light emission from local features are included) and the relatively close FPS results when compared to basic local illumination, it is clear that the first quality grouping should only be selected as a last resort effort to free up computational resources. Thus, the presented 3D engine will use the high quality lighting grouping, unless the number of light sources increases above 50 (especially taking into account the overall performance impact of additional rendering algorithms and other GPU burdens such as physics processing).

Running the same simulations with the light source count ranging from 55 to 65 shows a rapid decrease in the rendering frame rate. Figure 4.10 shows these results.

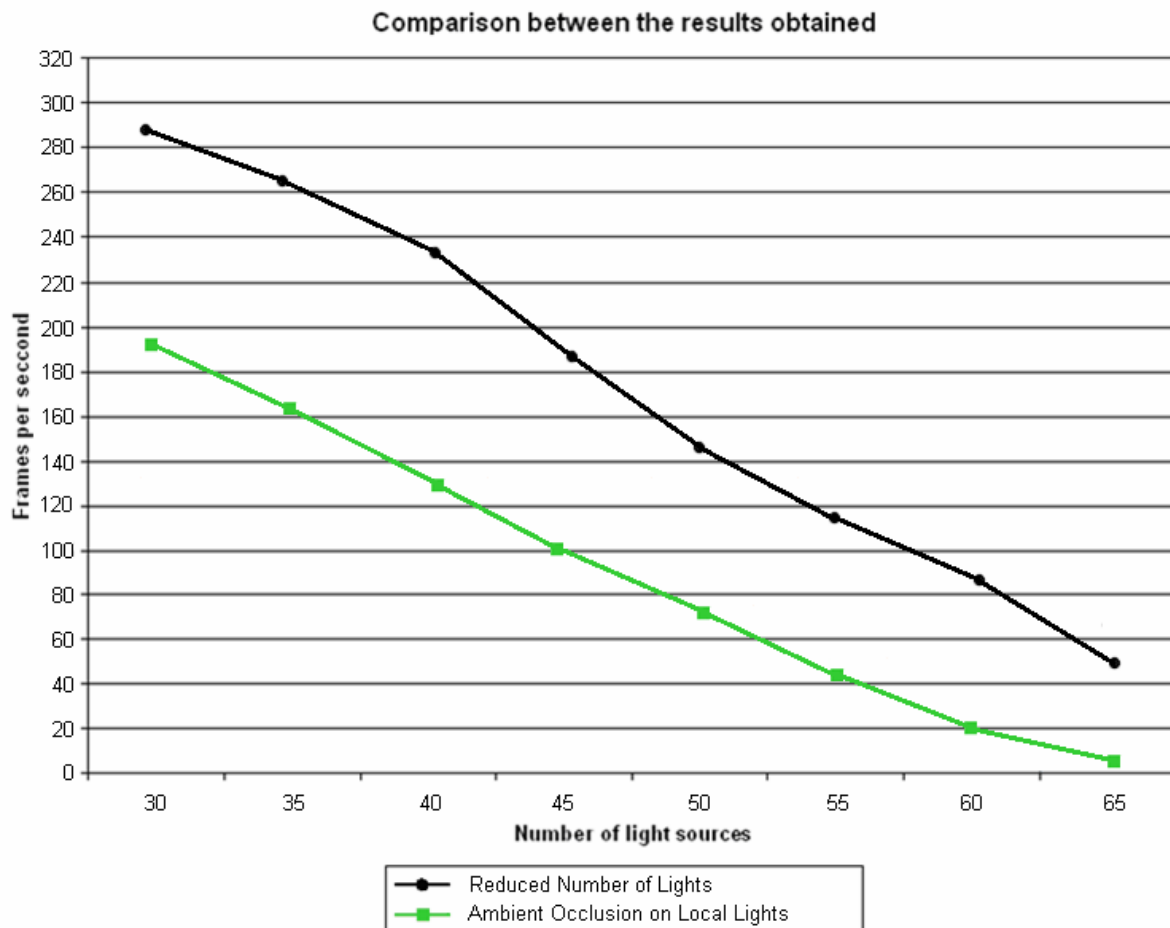


Figure 4.10 Comparison of all the previously listed quality scalings (30-65 light sources)

Lighting quality selection, as with shaders, is based on the optimisation of the rendering frame rate. Table 4.6 gives the most appropriate selection based on the presented algorithmic comparison.


Most Appropriate Selection	Conditions
Low Local Illumination.	The GPU is heavily overburdened and additional computational resources are required by other core rendering elements/the scene contains more than fifty light sources.
High Local Illumination.	The GPU is not fully utilised and the scene consists of fifty light sources or less and high-quality effects are required.

Table 4.6 Local illumination quality selections based on the presented critical analysis.

4.3.4 Reflection and Refraction

The presented rendering environment extends the basic local illumination lighting model by adding reflection and refraction effects to result in more realistic and lifelike images. When computation processing power is not available, the engine will utilise basic environmental mapping which allows us to simulate reflections by mapping real-time computed texture images to the surface of an object. Each texture image used for environmental mapping stores a “snapshot” image of the environment surrounding the mapped object. The engine further supports refractive environmental mapping, the Fresnel effect (Wloka, 2002) and chromatic dispersion resulting in an object’s colour being blended with reflections from its cube map (section 3.4). Thus, when the processing power is available, the presented renderer’s basic reflections can be extended to appear more lifelike.

The presented reflection quality evaluation focuses on a number of reflection and refraction implementations and approaches, specifically basic environmental mapping, CPU-based cube mapping, refractive environmental mapping and the extension of these reflection and refraction algorithms through the addition of the Fresnel effect and chromatic dispersion. Table 4.7 organises these implementation approaches into three reflection/refraction quality grouping: Low, Medium and High.

Grouping/Description	Rendered Scene Screenshot
<p>Low Reflection Quality</p> <p>Supports only GPU-based environmental mapping</p>	

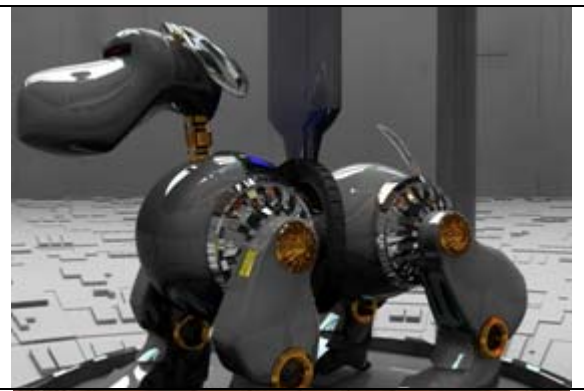

<p>Medium Reflection Quality</p> <p>Moves all environmental mapping computations off to the CPU (thus freeing the GPU in the process).</p>	
<p>High Reflection Quality</p> <p>Replaces basic environmental- or cube mapping with processor-intensive refractive environmental mapping supporting chromatic dispersion and the Fresnel effect.</p>	

Table 4.7 Reflection and refraction quality groupings.

Figures 4.11 and 4.12 give the observed performance of each reflection and refraction quality scaling group.

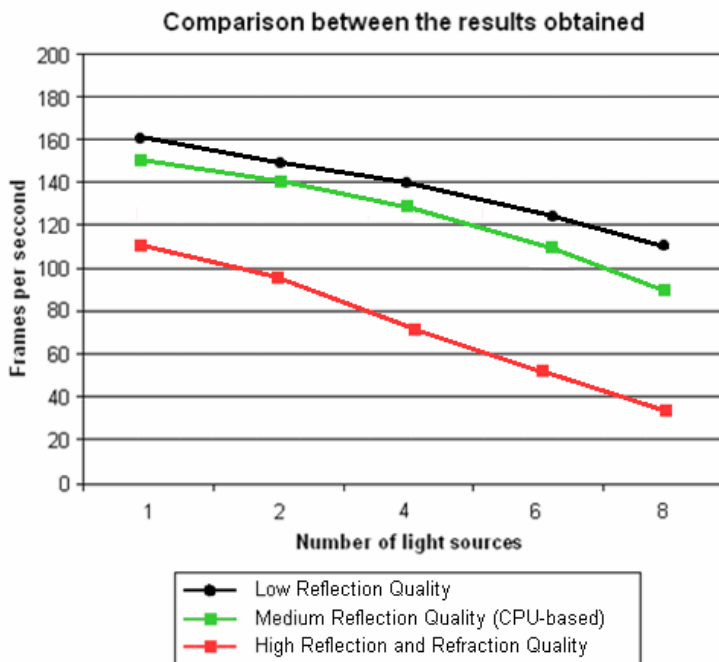


Figure 4.11 Comparison of all the previously listed quality scalings (1-8 light sources).

The first quality grouping, concerned with basic environmental mapping, is the best performing configuration and perfectly suited as a performance-orientated selection in situations where the GPU is being over-utilised or when faced with limited computational resources. As with the other groupings, this combination shows significant performance degradation when more light sources are added. It does, however, outperform all the remaining groups (the only cost being rendering quality).

The medium quality grouping performs only slightly worse than the first but given the fact that cube mapping is being performed on the CPU (thus freeing the GPU to perform other tasks) and that the frame rate is comparable to the same algorithm performed on the GPU, it is clear that CPU-based cube mapping is an excellent alternative to its only slightly better performing GPU-based counterpart.

The third grouping performs relatively well when dealing with scenes featuring one to eight light sources and when high-quality special effects are required. The presented renderer will only be utilising this grouping for scenes consisting of 7 light sources or less and where the processing resources are available to facilitate refractive environmental mapping, the Fresnel effect and chromatic dispersion. 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 4.12 shows these results.

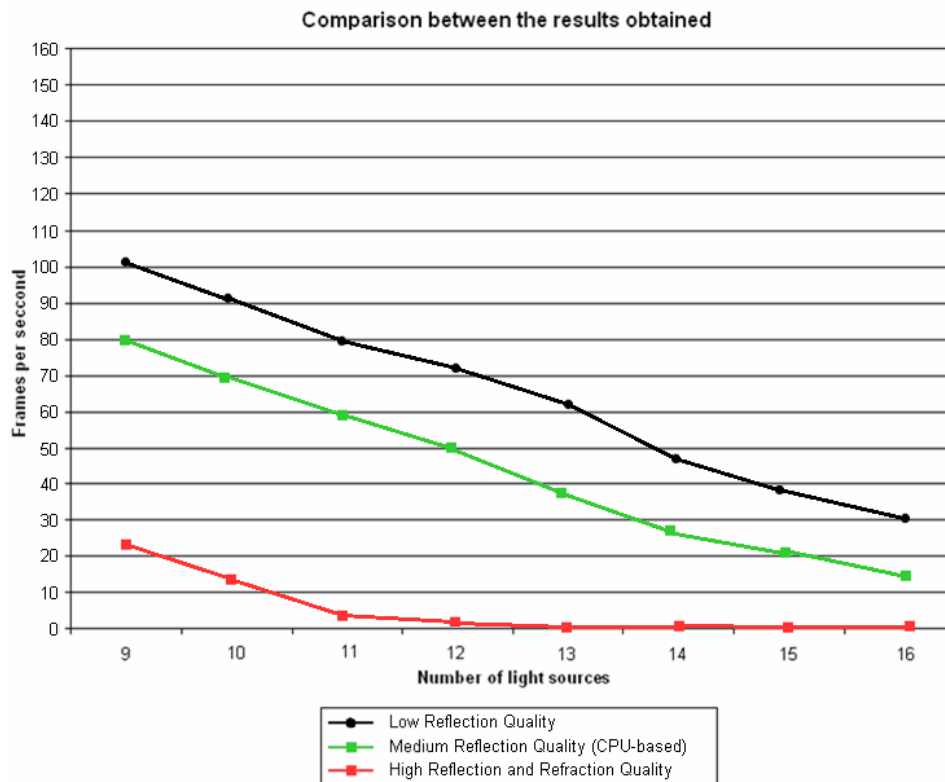


Figure 4.12 Comparison of all the previously listed quality scalings (9-16 light sources)

As with other shader implementations, reflection and refraction selection is based on the optimisation of the rendering frame rate and rendering quality. Also, rendering accuracy and reflection detail of distant objects (for instance, distant object reflections) carries less weight than those rendered relatively close to the viewer. Table 4.8 summarises the algorithms of choice based on the algorithmic comparison and scene conditions such as view distance, dynamic light conditions and number of light sources.

Most Appropriate Selection	Conditions
Low Reflection Quality.	The GPU is heavily overburdened, the CPU is fully utilised or cannot be utilised to lighten the GPU load and additional computational resources are required by other core rendering elements.
Medium Reflection Quality.	The GPU is fully utilised, additional computational resources are required and the CPU is not fully utilised and can be utilised to lighten the GPU load.
High Reflection and Refraction Quality.	The GPU is not fully utilised and the scene consists of one to seven light sources and high-quality special effects are required.

Table 4.8 Reflection and Refraction quality selections based on the presented critical analysis.

4.3.5 Physics

The presented rendering environment features not only basic physics simulations but also realistic object interaction based on Newton's Laws, a particle system inheriting from the physics system and realistic object interaction with all objects reacting based on the force exerted and environmental resistance. The engine's physics quality scaling (with quality relying on the GPU and/or CPU's computational power) is organised into performance-impacting selection categories ranging from Low to Very High. Specifically the categories which may be selected are either Off (very basic physics simulation), Low (75% Reduction in Physics Calculations), High (25% Reduction in Physics Calculations) or Very High (No Reduction in Physics Calculations).

The evaluation focuses on a number of physics calculations, specifically object acceleration, force, linear momentum, gravitational pull, projectile simulation through trajectory paths, friction and collision detection. The computational requirements for each of these are now presented with Figures 4.13 (a) and (b) giving the observed performance of each calculation group. However, Appendix E, in addition to section 3.7, can be consulted should background information be needed on the simulation of

Newtonian physics through the use of quantities such as mass, acceleration, velocity, friction, momentum, force, etc.

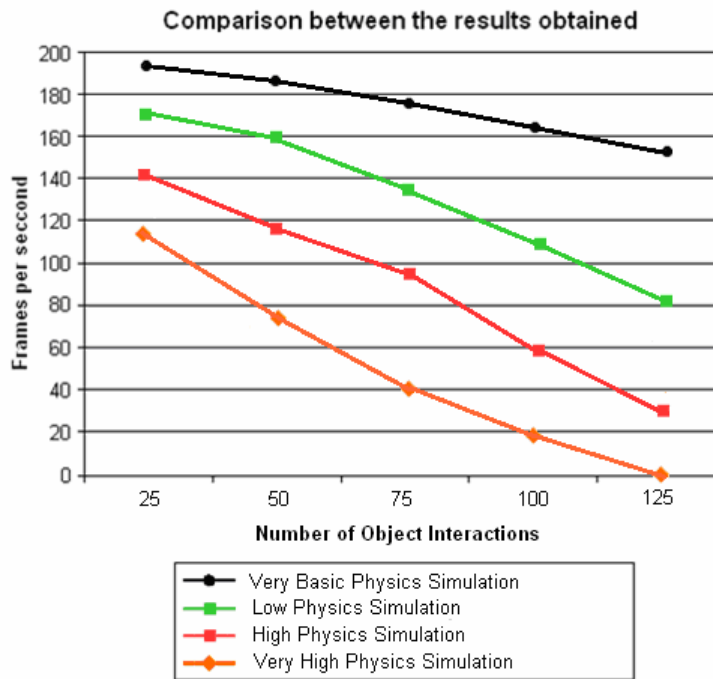


Figure 4.13 (a) Comparison of all the previously listed quality scalings – GPU (25-125 interacting objects).

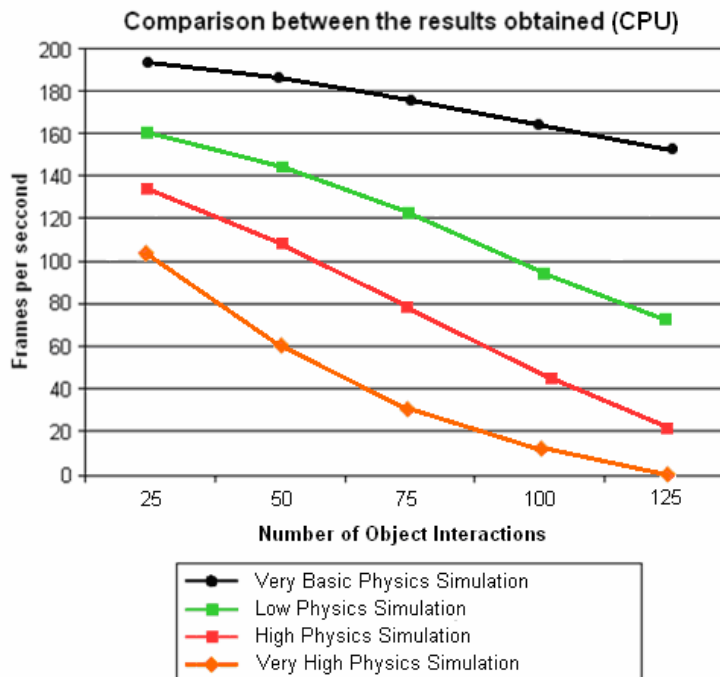


Figure 4.13 (b) Comparison of all the previously listed quality scalings – CPU (25-125 interacting objects).

Physics calculations performed on the GPU utilise NVIDIA's PhysX real-time physics engine (without any multithreading optimisation) while those performed on the CPU are based on the x87 floating point subset of the x86 architecture instruction set. CPU calculations fully utilise SSE multithreaded technology while PhysX calculations are not natively optimised for SSE or multithreading, resulting in situations where a PhysX implementation can be outperformed by well-coded multithreaded CPU physics implementations. There are thus two calculation subsets – the non-SSE PhysX code performed on the GPU and the multithreaded x87 code utilising SSE executed on the CPU. Simply running PhysX code (in software mode) on the CPU leads to significant performance drops (as thread control is to be handled by the developer).

The first quality selection, utilising only simple bounding boxes and basic edge detection and object interaction production rules, is the best performing configuration and perfectly suited as a performance-orientated selection in situations where the GPU and CPU are being over-utilised or when faced with limited computational resources. This selection shows a performance degradation as more objects are added. It does, not surprisingly, outperform all the remaining groups (the only cost being a lack in realism).

The low physics simulation selection performs, as expected, significantly worse than the first but given the quality benefits inherent to the utilisation of Newtonian physics (albeit with a reduction in computational accuracy), it is clear that the first quality selection should only be selected as a last resort effort to free up computational resources.

The third selection performs relatively well when dealing with scenes featuring up to 125 interacting objects and when highly accurate physics calculations are required. The presented rendering engine will only be utilising this grouping when highly accurate physics calculations are not possible. Very High Physics Simulations give proportionally lower performance figures and will only be utilised when the necessary computational resources are not required for graphics processing.

Physics selection is based on the optimisation of the rendering frame rate and rendering quality. The scene's frames per second performance data as well as the viewer's position in relation to the physics simulation are taken into account when selecting the most appropriate physics grouping. The accuracy of distant object simulations (for instance, two distant object colliding) will carry less weight than those rendered relatively close to the viewer. Table 4.9 summarises the groupings of choice based on our algorithmic comparison and scene conditions and number of objects.

Most Appropriate Selection	Conditions
Very Basic Physics Simulation.	The GPU is heavily overburdened, the CPU is fully utilised or cannot be utilised to lighten the GPU load and additional computational resources are required by other core rendering elements.
Low Physics Simulation.	The GPU is fully utilised, the CPU is fully utilised or cannot be utilised to lighten the GPU load but no additional computational resources are required.
High Physics Simulation.	The GPU or CPU is not fully utilised and the scene consists of one to 125 objects and high-quality physics are required but Very High Physics Simulation would overburden the GPU and/or CPU.
Very High Physics Simulation.	The necessary computational resources (CPU and/or GPU) are not required for graphics processing and the necessary physics calculations does not cause a noticeable drop in the perceivable smoothness of the scene being rendered.

Table 4.9 Physics quality selections based on the presented critical analysis.

4.3.6 Particle Effects

Particle effects, with the particle system inheriting from the physics system, allows for the simulation of natural phenomena such as fire, smoke, sparks, explosions, dust, trail effects, etc. As discussed in sections 3.7.2 and 3.7.3, the particle system is implemented using three stages, namely, the setup stage, the simulation stage and the rendering stage.

The setup stage involves specification of the particle system's spatial position and area of constraint – parameters controlled by the emitter. The emitter also controls the particle creation rate, that is, the rate at which new particles are injected into the system. Each particle has a specific time to live, after which it is destroyed.

The simulation stage takes care of particle rendering rates, particle spawning position (mostly randomised between some minimum and maximum coordinate range), particle properties (such as particle colour, velocity, etc) and positioning of the emitter. This stage also keeps track of each particle to check whether a specific particle has exceeded its lifetime. Each particle has an initial velocity and is translated based on

some sort of physics model or simply by adding velocity to its current spatial position. Collision detection is also possible at this stage but rarely implemented.

Following the simulation state, each particle is rendered as either a coloured point, polygon or as a mesh.

The engine’s particle system, based on the rules of physics, uses the following standard equations to calculate each particle’s velocity and position:


$$V_{new} = V_{old} + a \times t$$

$$Pos_{new} = Pos_{old} + (v_{old} \times t) + \left(\frac{1}{2} \times a \times t^2\right),$$

The above given equations factor in the initial motion of the particle, its trajectory and the overall effect of gravity where

- Pos_{new} is the particle’s final position,
- Pos_{old} its initial position,
- V_{new} its final velocity,
- V_{old} its initial velocity,
- a the particle’s acceleration and
- t the change in time.

Using these equations we start by initialising each particle’s initial position and velocity. These values will be assigned to a particle when it is generated by the emitter. The quality scaling (with quality of explosions, dust, tread marks, beams, etc relying on the CPU and GPU’s computational power) of the rendering engine’s particle effects is, as with physics, organised into performance-impacting selections ranging from Low to Very High. Table 4.10 lists these particle effects scaling approaches with Figures 4.14 (a) and (b) giving the mean performance of each calculation group as executed on the CPU and GPU, respectively.

Grouping/Description	Rendered Scene Screenshot
<p>Low Particle Simulation</p> <p>75% reduction in effect quality.</p>	




<p>Medium Particle Simulation</p> <p>50% reduction in effect quality.</p>	
<p>High Particle Simulation</p> <p>25% reduction in effect quality.</p>	
<p>Very High Particle Simulation</p> <p>No reduction in effect quality.</p>	

Table 4.10 Particle effects quality groupings.

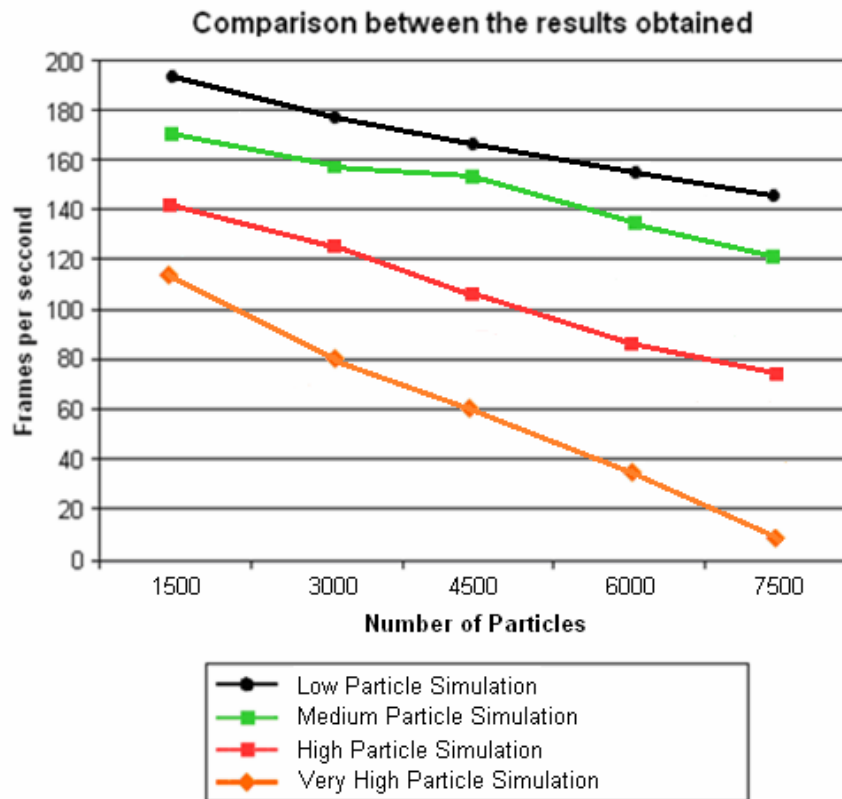


Figure 4.14 (a) Comparison of all the previously listed quality scalings – GPU

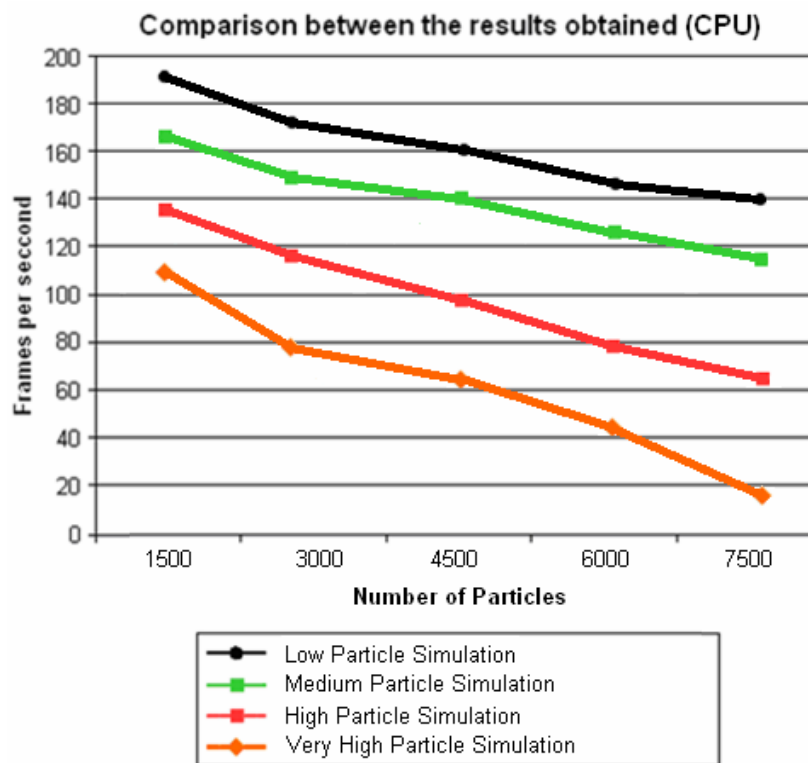


Figure 4.14 (b) Comparison of all the previously listed quality scalings – CPU.

As with physics, particle calculations – velocity and position – performed on the GPU utilises NVIDIA’s PhysX real-time physics engine while those performed on the CPU are based on the x87 floating point subset of the x86 architecture instruction set.

The first quality selection, representing a 75% reduction in effect quality, is the best performing configuration and perfectly suited as a performance-orientated selection in situations where the GPU and CPU are being over-utilised or when faced with limited computational resources. As with the other performance selections, this combination shows a performance degradation as the number of particles increase. Unsurprisingly, it outperforms all the remaining selections (the only cost being a lack in visual quality and realism).

The medium particle simulation selection performs, as expected, slightly worse than the first but given the quality benefits inherent to the utilisation of more accurate Newtonian physics (albeit with a reduction in computational accuracy), it is clear that the first quality selection should only be selected as a last resort effort to free up computational resources.

The third selection performs relatively well when dealing with effects consisting of 1500 to 7500 particles and when highly accurate physics calculations are required. The presented 3D engine will only be utilising this selection when very high particle simulations are not possible. The final selection gives proportionally lower performance figures and will only be utilised when the necessary computational resources are not required for graphics processing.

Table 4.11 summarises the algorithms of choice based on our algorithmic comparison, scene conditions and number of particles.


Most Appropriate Selection	Conditions
Low Particle Simulation.	The GPU is heavily overburdened, the CPU is fully utilised or cannot be utilised to lighten the GPU load and additional computational resources are required by other core rendering elements.
Medium Particle Simulation.	The GPU is fully utilised, the CPU is fully utilised or cannot be utilised to lighten the GPU load but no additional computational resources are required.
High Particle Simulation.	The GPU or CPU is not fully utilised and the effect consists of 1500 to 7500 particles and high-quality physics are required but Very High Particle Simulation would overburden the GPU and/or GPU.

Very High Particle Simulation.	The necessary computational resources (CPU and/or GPU) are not required for graphics processing and the necessary Newtonian calculations does not cause a noticeable drop in the perceivable smoothness of the scene being rendered.
--------------------------------	--

Table 4.11 Effect quality selections based on our critical analysis.

4.3.7 Post-Processing

The presented rendering engine uses post-processing or quality-improvement image processing (through the use of pixel shaders) to add additional effects such as bloom lighting (the effect of producing light fringes around ultra-bright objects – an object with a bright light behind it will be “overlapped” by the light and thus appear more lifelike), motion blur (the streaking of rapid moving objects), ambient occlusion (the global effect of approximating the radiation of light by the casting of rays in every direction from an object’s surface), depth of field (the variance in sharpness between the nearest and farthest objects in a scene), displacement mapping (as an alternative to normal mapping – used to displace surface points; giving surfaces great depth and detail) and halo effects (artificial glow added to light “emitting” objects such as light bulbs or a glowing red button). The engine’s post-processing quality scaling relies on the GPU’s computational power and consists of three quality groups: Low, Medium and High. Table 4.12 lists these scaling approaches with Figure 4.15 showing the mean performance of each post-processing quality scaling group.

Grouping/Description	Rendered Scene Screenshot
<p>Low Post-Processing</p> <p>Adds Minimal Intensity Bloom Effects and Displacement Mapping to the rendered scene.</p>	

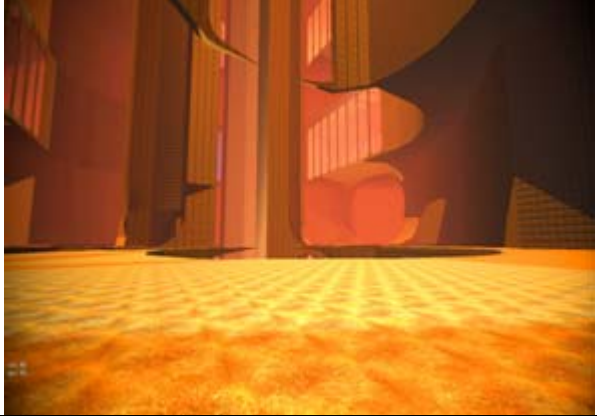

<p>Medium Post-Processing</p> <p>Adds Ambient Occlusion (along with High Intensity Bloom Effects and Displacement Mapping).</p>	
<p>High Post-Processing</p> <p>Adds Depth of Field and Halo Effects to the rendered scene.</p>	

Table 4.12 Post-Processing effects quality groupings.

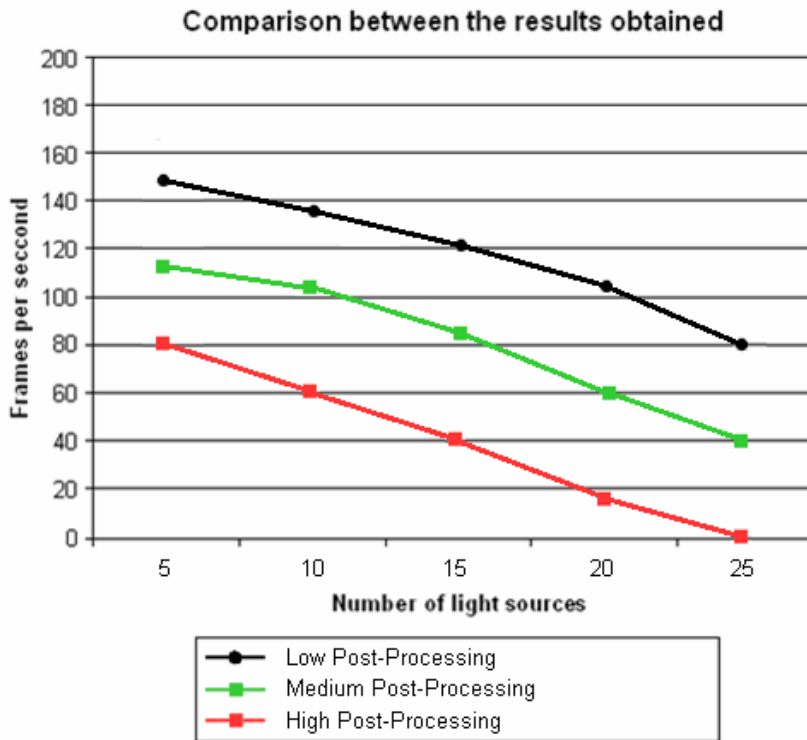


Figure 4.15 Comparison of all the previously listed quality scalings (5-25 light sources).

The first quality grouping, consisting of displacement mapping and bloom effects, is the best performing configuration and is well suited as a performance-orientated selection in situations where the GPU is being over-utilised or when faced with limited computational resources. As with the other groupings, this combination shows significant performance degradation as more light sources are added. It does, however, outperform all the remaining groups (the cost is not so much a loss in rendering detail as it is one where there are “less” special effects than when using the others; this selection simply excludes ambient occlusion, depth of field and halo effects).

The medium post-processing quality grouping performs somewhat worse than the first but given the global lighting benefits inherent to the utilisation of ambient occlusion, it is clear that the first quality grouping should only be selected as a last resort effort to free up computational resources.

The final grouping performs relatively well when dealing with scenes featuring one to ten light sources and when high-quality special effects are required. This grouping will be utilised for scenes consisting of 15 light sources or less and where the processing resources are available to facilitate ambient occlusion, depth of field, displacement mapping and halo and bloom effects.

The presented rendering engine selects the most appropriate grouping given the processing power available. Also, the rendering accuracy and detail of distant objects (for instance, distant displacement mapping or bloom calculations) carry less weight than those rendered relatively close to the viewer. Table 4.13 summarises the algorithms of choice based on our algorithmic comparison and scene conditions such as view distance, dynamic/static light conditions and number of light sources.

Most Appropriate Selection	Conditions
Low Post-Processing.	The GPU is heavily overburdened and additional computational resources are required by other core rendering elements.
Medium Post-Processing.	The GPU is fully utilised but no additional computational resources are required.
Very High Post-Processing.	The scene contains less than fifteen light sources and the computational resources are available to facilitate ambient occlusion, depth of field, displacement mapping and halo and bloom effects.

Table 4.13 Post-Processing quality selections based on our critical analysis.

4.4 Summary

In the chapter we started by presenting a benchmarking mechanism and a set of criteria for the evaluation of rendering algorithms and techniques. The given evaluation criteria were selected with the aim of assessing the relationship between rendering quality and performance – in turn allowing for, where applicable, the isolation of key algorithmic weaknesses and possible bottleneck areas.

Specific shadow 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 Rautenbach et al's algorithm based on shadow volumes, spatial subdivision and instruction set utilisation.

The subsequent shader evaluation, in turn, focused on a number of shader implementations and lighting approaches, after which two local illumination configurations were investigated. The first of these limiting the number of light sources in an attempt to reduce GPU utilisation with the second lifting this limitation while occluding local light sources (a technique used to approximate the effect of environment lighting as an attempt to simulate the way light radiates in real life). This evaluation was later extended with the inclusion of HDR lighting.

Next the evaluation focused on a number of reflection and refraction implementations and approaches, specifically: basic environmental mapping, CPU-based cube mapping, refractive environmental mapping and the extension of these reflection and refraction algorithms through the addition of the Fresnel effect and chromatic dispersion.

The chapter then shifted focus to the evaluation of a number of physics calculations such as object acceleration, force, linear momentum, gravitational pull, projectile simulation through trajectory paths, friction and collision detection followed by the benchmarking of our engine's dynamically allocated particle generator.

Chapter 4 concluded with the performance analysis of a number of post-processing shader implementations and lighting approaches, specifically displacement mapping, bloom effects, ambient occlusion, depth of field and halo effects.

An Empirically Derived System for Distributed Rendering

Chapter 5 presents our empirically derived system for distributed rendering. 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:

- Dynamic algorithm selection
- Rules for selection of rendering algorithms
- Fuzzy rules for selection of the most appropriate rendering algorithm
- Construction of the algorithm selection mechanism
- Results obtained from our benchmarking environment

5.1 Introduction

The presented real-time rendering engine continuously analyses a dataset to determine the best solution to a given rendering problem – as in, the best algorithm or shader to use for a specific scene or a specific object in a scene. This selection system consists of an empirically ascertained dataset (containing the previously obtained algorithmic performance data), a collection of rules to analyse the data and information of various elements pertaining to the scene currently being rendered.

The rendering engine uses a selection engine to control the real-time selection of rendering algorithms and, when performing cube mapping or physics calculations, to effectively distribute processing between the CPU and GPU. The knowledge base of this engine, consisting of production rules, is derived from experimental results obtained through the critical analysis of numerous real-time rendering algorithms, as discussed in Chapter 4. These production rules are used by an inference engine which, in turn, is tasked with the selection of the most appropriate algorithm based on certain properties of the scene being rendered. For instance, the presented 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 (Salton, 1987). 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. The presented engine combines production rules with fuzzy logic to explicitly symbolise data. This is followed by the selection of the most appropriate rendering algorithm.

The next section presents this selection engine implementation in detail. Following this, a critical analysis of the empirically derived system for the high-speed rendering of complex 3D environments is performed. This analysis will convey not only the performance benefits inherent in the utilisation of this system, but also the practicality of such an implementation.

5.2 The Selection Engine and the Dynamic Selection and Allocation of Algorithms

The empirically derived system for high-speed rendering consists of a fuzzy logic based selection engine and several rendering algorithms and approaches. The selection engine controls, as mentioned, the selection and allocation of these algorithms by

correlating the properties of the scene being rendered with the previously obtained algorithmic performance data.

The selection engine consists of the following modules:

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

The selection engine's knowledge base consists of experimental results obtained through the critical analysis of numerous real-time rendering algorithms. The inference engine is in turn used to select the most appropriate algorithm based on certain properties of the scene being rendered. The knowledge base is nothing more than a database of rules. These rules symbolise the stored knowledge. The fact database embodies the selection engine inputs (properties and performance statistics of the scene being rendered) which are subsequently used to make decisions and/or to take certain actions. The inference engine makes the actual decision by combining these selection engine rules and facts. The explanation system, implemented only in skeletal form should future developers require debugging information, generates information about the manner in which a decision was made. Figure 5.1 illustrates the architecture of the presented selection engine.

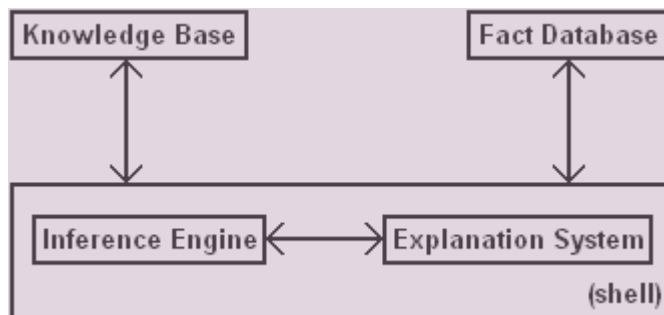


Fig 5.1 Architecture of the selection engine.

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

The selection engine implementation utilises a forward chaining strategy to determine results from a collection of rules and facts. The process basically starts by reading the selection engine 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.

As mentioned in Chapter 4, the selection and/or allocation of algorithms is based on the continuous optimisation of the rendering frame rate and overall rendering quality. The implemented selection engine will thus select the most appropriate algorithms by taking not only the scene's frames per second performance data (dynamically changing as one moves through the scene) into account but also by factoring in the viewer's position in relation to the object or effect being rendered or calculated. The rendering accuracy and detail of distant objects or effects will thus carry less weight than those rendered relatively close to the viewer. The next section presents the dynamic selection and allocation (where applicable) of the algorithms and rendering approaches discussed in Chapter 3.

5.2.1 Shadows

The following rules can be defined for selecting the most appropriate shadow rendering algorithm (these rules are derived from the algorithmic comparison given in Chapter 4 – Table 4.4 summarises the shadow algorithms of choice based on this algorithmic comparison as well as scene conditions such as view distance, dynamic/static light conditions and number of light sources):

Rule #1

If the environment/sub-environment consists of only static light sources,
Then render all shadows via Rautenbach et al's spatial subdivision/SSE2 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 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 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 algorithm.

An interesting aspect of the presented selection engine implementation is its fuzzy logic-based nature. As a fuzzy logic based selection engine, 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 applies a set of fuzzy rules on a set of traditional precise inputs to obtain a precise output value (such as an action recommendation).

The presented fuzzy logic based selection engine will thus contain the following redefined, "fuzzified" 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 Rautenbach et al's spatial subdivision/SSE2 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 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 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 algorithm.

Mamdani inference is used to “fuzzify” all precise input values via the definition of fuzzy sets. Here 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). The presented 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).

The presented 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. For a thorough evaluation of these rules, please see the MSc dissertation, *An Empirically Derived System for High-Speed Shadow Rendering* (2008).

5.2.2 Shaders

As in the previous section, we can define several rules for the selection of the most appropriate shader quality scaling (section 4.3.2 presents the shader comparison, based on scene conditions such as view distance and the number of light sources, upon which these rules are based):

Rule #1

If the GPU is heavily overburdened,
And additional computational resources are required by other core rendering elements
Then render the scene using the Low Shader Quality grouping.

Rule #2

If the GPU is fully utilised

And no additional computational resources are required

Then render the scene using the Medium Shader Quality grouping.

Rule #3

If the GPU is not fully utilised,

And the scene consists of a few or less than a below average number of light sources,

And high-quality special effects are required,

And Very High Shader Quality would overburden the GPU,

Then render the scene using the High Shader Quality grouping.

Rule #4

If the scene contains a below average number of light sources

And the computational resources are available to facilitate true HDR lighting, translucent shadows, parallax mapping and volumetric materials,

Then render the scene using the Very High Shader Quality grouping.

5.2.3 Local Illumination

The following fuzzy rules, based on the comparison given in section 4.3.3, deal with the dynamic selection of the most appropriate local illumination quality approach:

Rule #1

If the GPU is heavily overburdened,

And additional computational resources are required by other core rendering elements

Or the scene contains a great number of light sources

Then render the scene using the Low Local Illumination grouping.

Rule #2

If the GPU is not fully utilised,

And the scene contains less than a very high number of light sources,

And high-quality special effects are required,

Then render the scene using the High Local Illumination grouping.

5.2.4 Reflection and Refraction

The most appropriate reflection and refraction quality approaches, as presented in section 4.3.4, can now be selected in real-time using the following fuzzy rules:

Rule #1

If the GPU is heavily overburdened,
And the CPU is fully utilised
Or cannot be utilised to lighten the GPU load,
And additional computational resources are required by other core rendering elements
Then render the scene using the Low Reflection Quality grouping.

Rule #2

If the GPU is fully utilised,
And additional computational resources are required,
And the CPU is not fully utilised,
And can be utilised to lighten the GPU load,
Then render the scene using the Medium Reflection Quality grouping.

Rule #3

If the GPU is not fully utilised,
And the scene consists of a few or less than a below average number of light sources,
And high-quality special effects are required,
Then render the scene using the High Reflection and Refraction Quality grouping.

5.2.5 Physics

The following fuzzy rules control the selection of the most appropriate physics simulation approach based on the presented algorithmic comparison (section 4.3.5):

Rule #1

If the GPU is heavily overburdened,
And the CPU is fully utilised
Or cannot be utilised to lighten the GPU load,
And additional computational resources are required by other core rendering elements
Then implement physics using the Very Basic Physics Simulation grouping.

Rule #2

If the GPU is fully utilised Or If the CPU is fully utilised Or cannot be utilised to lighten the GPU load,
And no additional computational resources are required,
Then implement physics using the Low Physics Simulation grouping.

Rule #3

If the GPU is not fully utilised Or If the CPU is not fully utilised
And the scene consists of less than an above average number of objects,
And high-quality physics are required,
 And Very High Physics Simulation would overburden the GPU and/or GPU,
Then implement physics using the High Physics Simulation grouping.

Rule #4

If the necessary computational resources (CPU and/or GPU) are not required for graphics processing,
And the necessary physics calculations does not cause a noticeable drop in the perceivable smoothness of the scene being rendered,
Then render the scene using the Very High Physics Simulation grouping.

5.2.6 Particle Effects

The most appropriate particle simulation selection (controlled using scene conditions and number of particles, as noted in section 4.3.6) is determined using the following set of rules:

Rule #1

If the GPU is heavily overburdened,
And the CPU is fully utilised
 Or cannot be utilised to lighten the GPU load,
And additional computational resources are required by other core rendering elements
Then implement particle effects using the Low Particle Simulation grouping.

Rule #2

If the GPU is fully utilised
And the CPU is fully utilised
 Or cannot be utilised to lighten the GPU load,
And no additional computational resources are required,
Then implement particle effects using the Medium Particle Simulation grouping.

Rule #3

If the GPU is not fully utilised Or If the CPU is not fully utilised
And the effect consists of a medium to high number of particles,
And high-quality physics are required,

And Very High Particle Simulation would overburden the GPU and/or GPU,
Then implement physics using the High Physics Simulation grouping.

Rule #4

If the necessary computational resources (CPU and/or GPU) are not required for graphics processing,

And the necessary Newtonian calculations does not cause a noticeable drop in the perceivable smoothness of the scene being rendered,

Then render the scene using the Very High Particle Simulation grouping.

5.2.7 Post-Processing

Table 4.13 summarises the algorithms of choice based on our algorithmic comparison and scene conditions such as view distance, dynamic/static light conditions and number of light sources. Further defining our selection engine, we can create the following fuzzy rules for selection of the most appropriate post-processing quality approach:

Rule #1

If the GPU is heavily overburdened,

And additional computational resources are required by other core rendering elements

Then render the scene using the Low Post-Processing Quality grouping.

Rule #2

If the GPU is fully utilised

And no additional computational resources are required

Then render the scene using the Medium Post-Processing Quality grouping.

Rule #3

If the GPU is not fully utilised,

And the computational resources are available to facilitate ambient occlusion, depth of field, displacement mapping and halo and bloom effects

And the scene consists of a less than average number of light sources,

Then render the scene using the Very High Post-Processing Quality grouping.

5.3 Construction of the Algorithm Selection Mechanism

The performance data gathered during the previously discussed critical analysis allows for the construction of a fuzzy logic-based selection and allocation system. This system, as mentioned, controls the real-time selection of rendering algorithms and quality

groupings based on environmental conditions. The gathered data (algorithm, shader and rendering performance) is stored in a comma-delimited format with the rendering engine loading it into memory via the in-game loop (section 2.2) upon execution. Each implemented algorithm/rendering approach is, in turn, loaded into the engine 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. For example, 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 Rautenbach et al's algorithm based on shadow volumes, spatial subdivision and instruction set utilisation.

CPU utilisation monitoring is performed using Intel's CPUUsage class (Intel, 2010). This class, wrapping Microsoft's Performance Data Helper (PDH) API (used to collect performance data of various performance counters or system instances), provides an interface for the calculation of maximum, minimum and average CPU utilisation over a period of time. As stated by Intel (2010), CPU utilisation is a key metric for optimisation, performance analysis, and workload evaluation. However, the built-in Windows facilities for tracking CPU utilisation provide limited flexibility. The CPUUsage class attempts to alleviate this issue by providing a simple interface that can be used to programmatically track CPU percentage. The level of control provided by the CPUUsage class allows virtually unlimited CPU utilisation monitoring options for the application developer.

NVIDIA's PerfKit (and the PerfSDK API) is, in turn, used by the rendering engine to access the physical GPU hardware counters and GPU usage data in real-time. The NVPerfKit is actually a collection of performance monitoring, debugging and profiling utilities focused on accessing the low-level performance indicating components of the graphics driver and the GPU itself (assuming an NVIDIA GPU is being used). These low-level components are known as performance counters. They give information on the application's overall frames per second rendering, the video memory used in MB, the graphics driver's sleep time, the polygon count, etc (NVIDIA, 2011). Using the NVPerfKit, we are thus able to profile the rendering engine in terms of its GPU, driver and memory usage. A useful component of NVPerfKit is called PerfHUD, a real-time Direct3D and OpenGL application profiler that generates its output in the form of a heads-up display (shown in Figure 5.2).

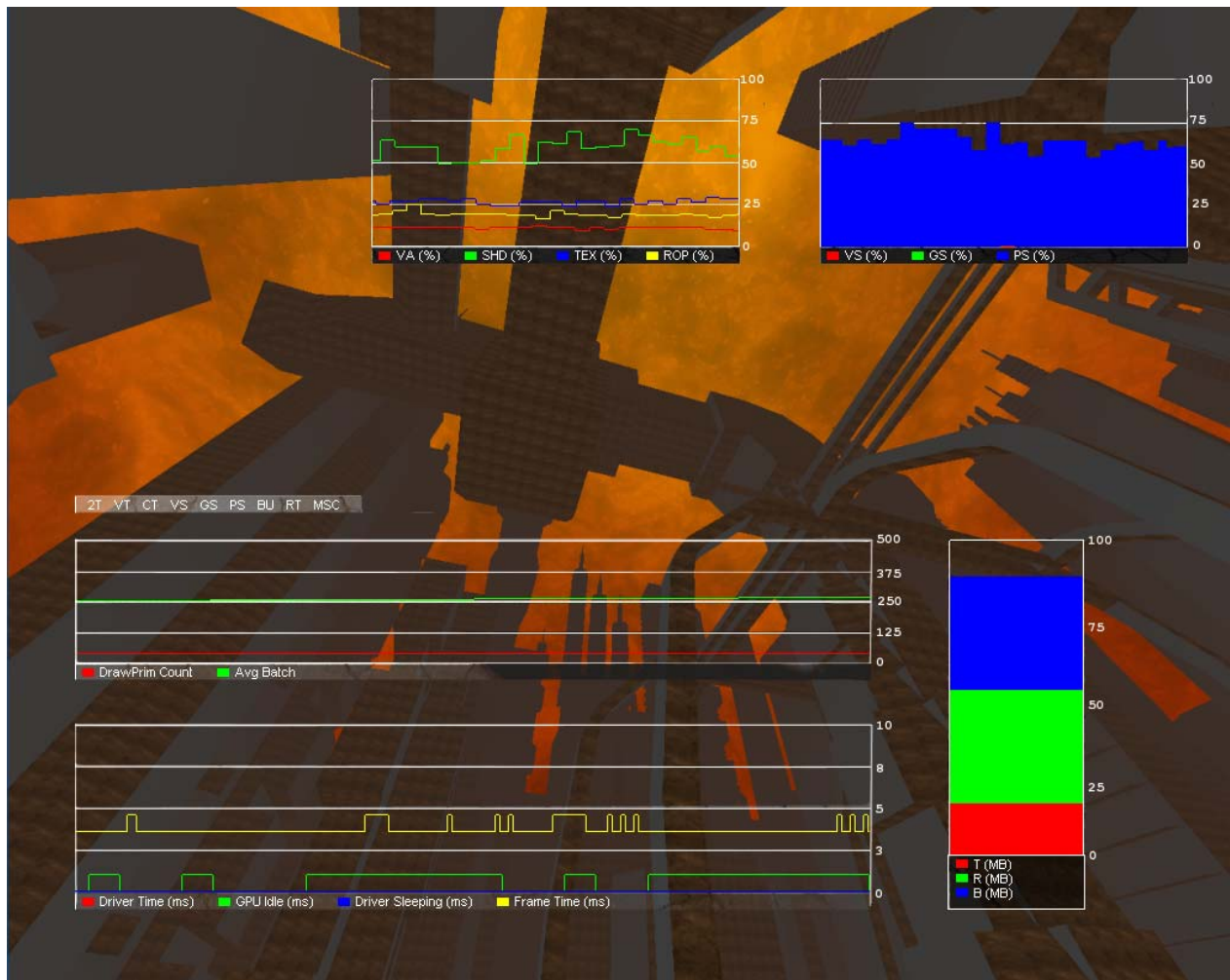


Figure 5.2 Nvidia's PerfHUD.

5.4 Results

By dynamically cycling through algorithms and quality groupings to compensate for performance-impacting changes in the presented rendering environment, we are able to bridge an existing gap between quality and high-speed rendering. The performance gains inherent in this system's use, when compared to traditional implementations, is subsequently highlighted.

We now describe the behaviour of the rendering engine when subjected to different scenarios, each performed independently of one another, that were designed to test its transition behaviour in respect of the various transition rules described above. The collective overall effect is a highly optimised rendering engine (the accompanying CD contains a high-definition video showcasing the rendering engine and the combined overall effect of dynamic quality selection and process allocation). Figure 5.3 shows a collage of the rendering engine in action.

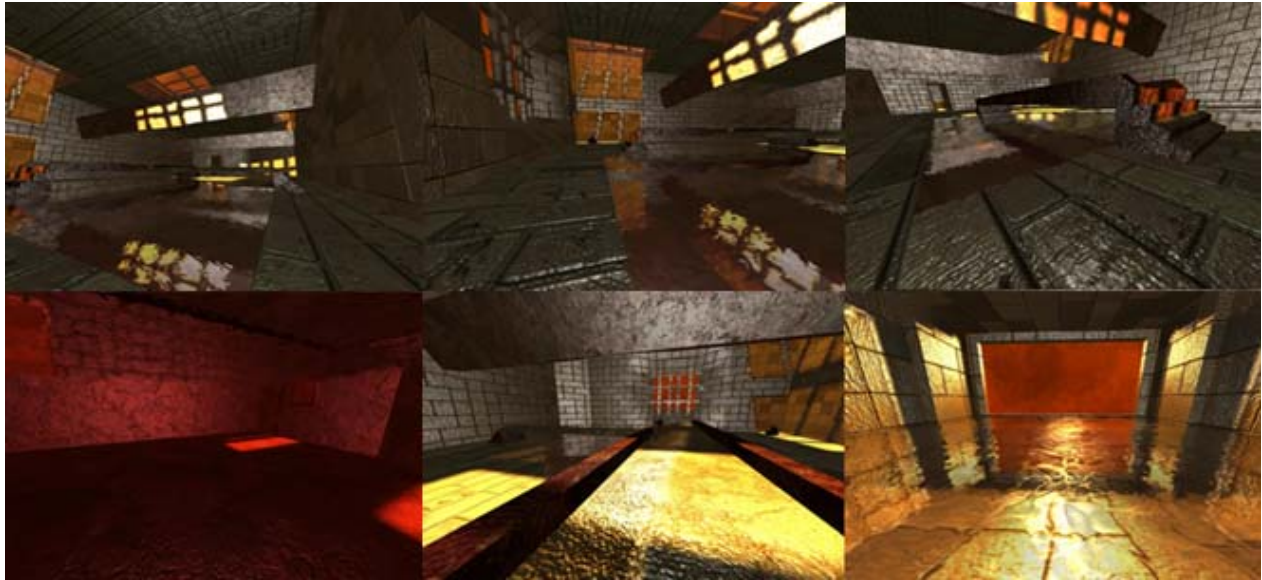


Figure 5.3 Various screenshots of the presented 3D engine.

Shadows

Starting with the engine's shadow quality scaling, as discussed in Rautenbach (2008), the presented benchmarking environment initially consisted of a 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 algorithm to Chan and Durand's 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 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 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 algorithm was successfully selected.

Figure 5.4 shows the performance data obtained (for this specific instance) for up to eight light sources with Figure 5.5 showing the results obtained for nine to sixteen light sources.

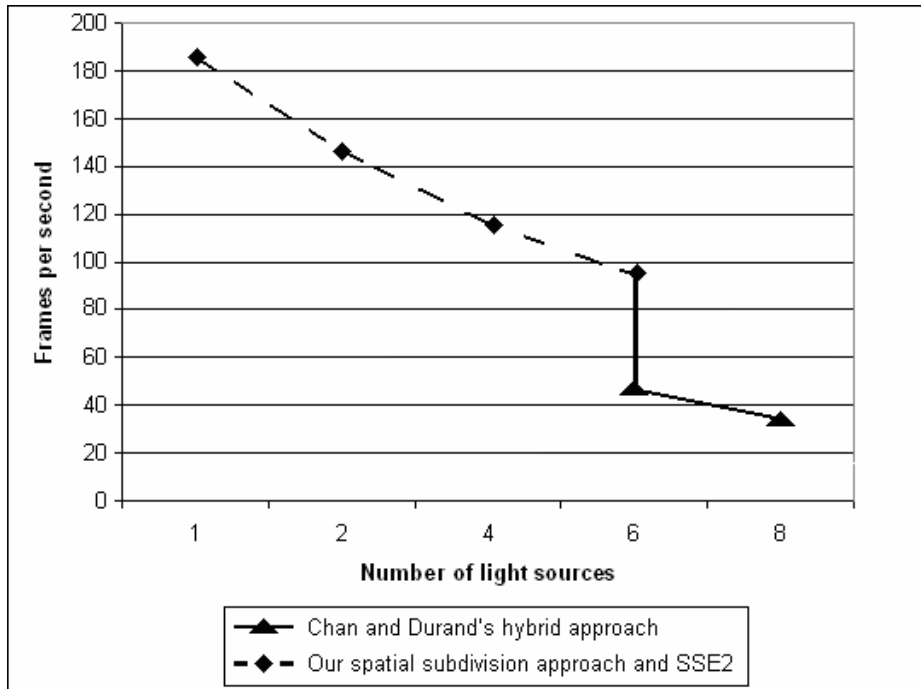


Figure 5.4 Shadow performance data for up to eight light sources.

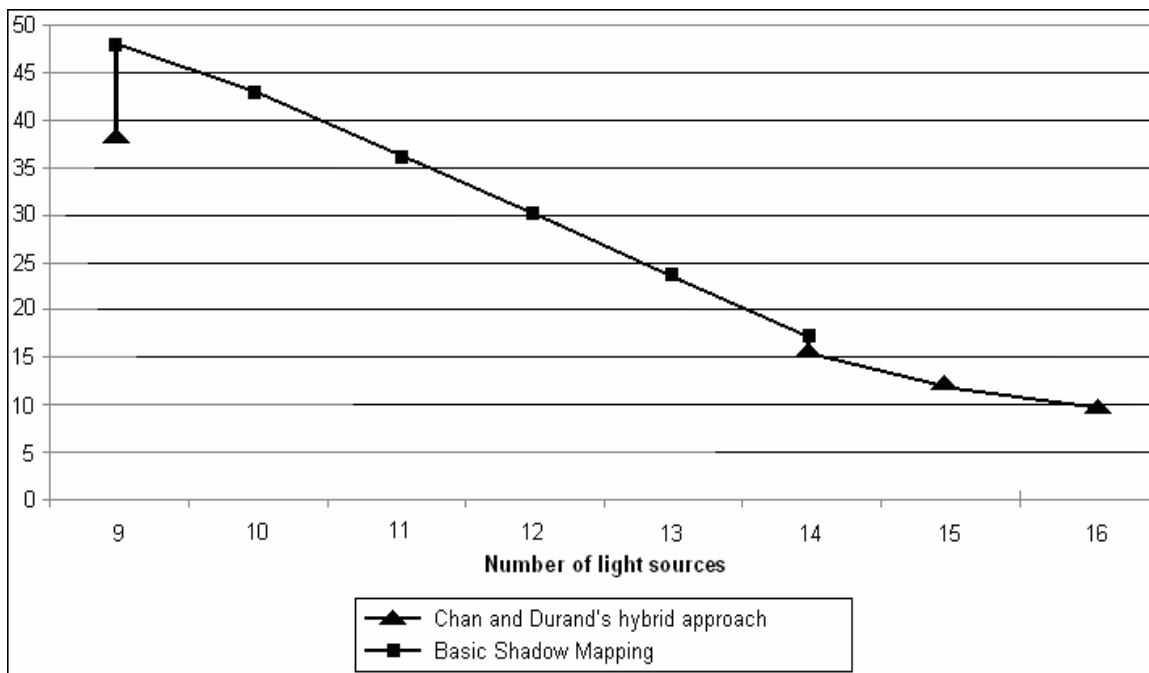


Figure 5.5 Shadow performance data for nine to sixteen light sources.

The experiment can be repeated 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 selected Chan and Durand’s 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 algorithm was still the algorithm of choice and no alternative shadow rendering algorithms was 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 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 algorithm to the spatial subdivision/SSE2 algorithm.

Our final action was to set all the dynamic light sources to static. Figure 5.6 shows the performance data obtained for sixteen to nine light sources with Figure 5.7 showing the results obtained for eight to a single light source.

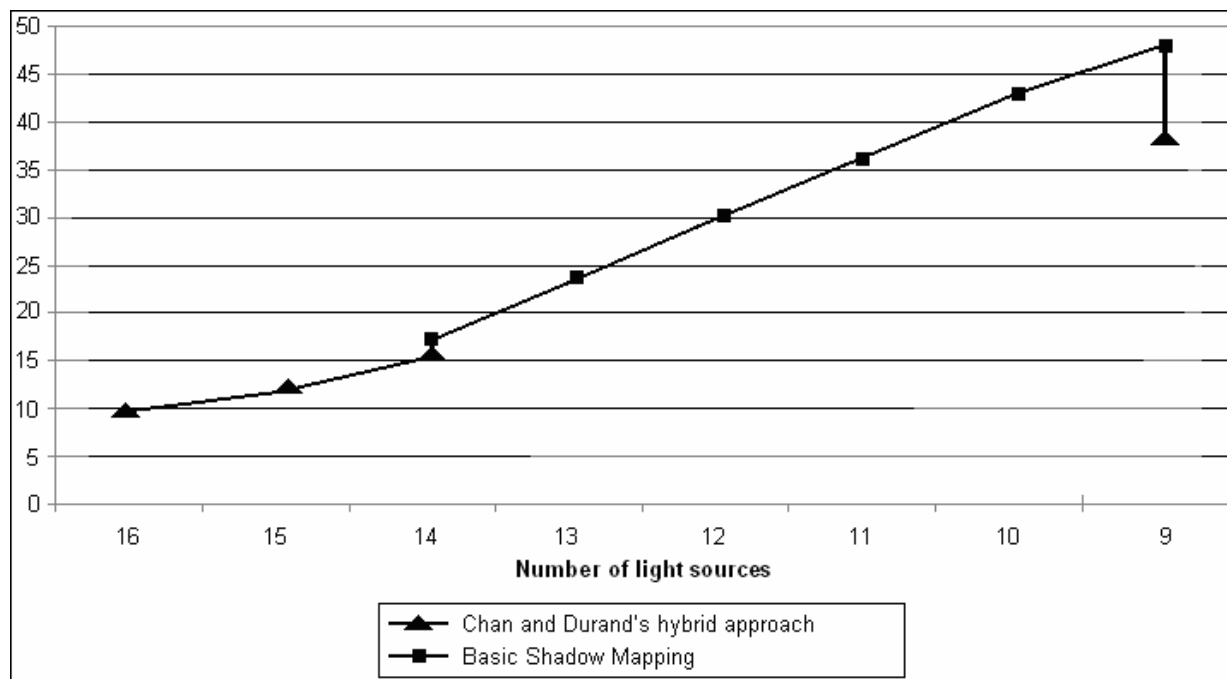


Figure 5.6 Shadow performance data for sixteen to nine light sources.

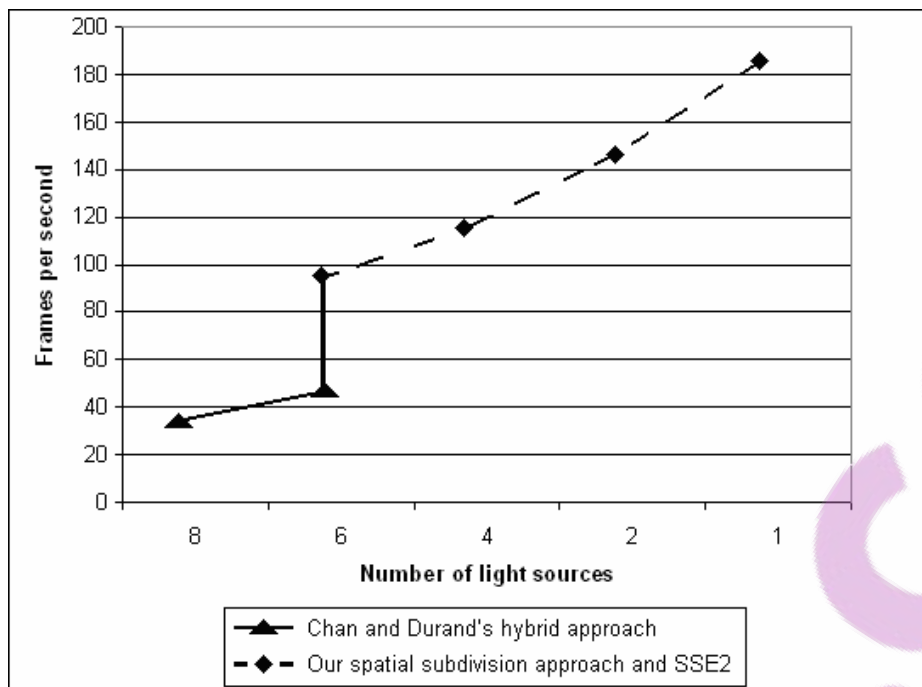


Figure 5.7 Shadow performance data for eight to a single light source.

Shaders

Similarly, for shader quality scaling, the presented benchmarking environment initially consisted of a single light source. We then added seven additional light sources with a number of objects positioned relatively close to the viewer. This allowed us to analyse the transition from the Very High Shader Quality grouping to the High Shader Quality Grouping.

Following this we increased the number of dynamic light sources to thirteen. The scene previously rendered using the High Shader Quality grouping were now rendered using simplified High Dynamic Range Lighting, normal maps, specular highlights and volumetric fog (the Medium Quality Grouping).

Next we systematically increased the number of light sources to sixteen. This caused a selection of the Low Shader Quality grouping.

The scene's lighting was now reset to seven dynamic light sources – the Very High Shader Quality grouping was successfully selected.

Figure 5.8 shows the performance data obtained (for this specific instance) for up to eight light sources with Figure 5.9 showing the results obtained for nine to sixteen light sources.

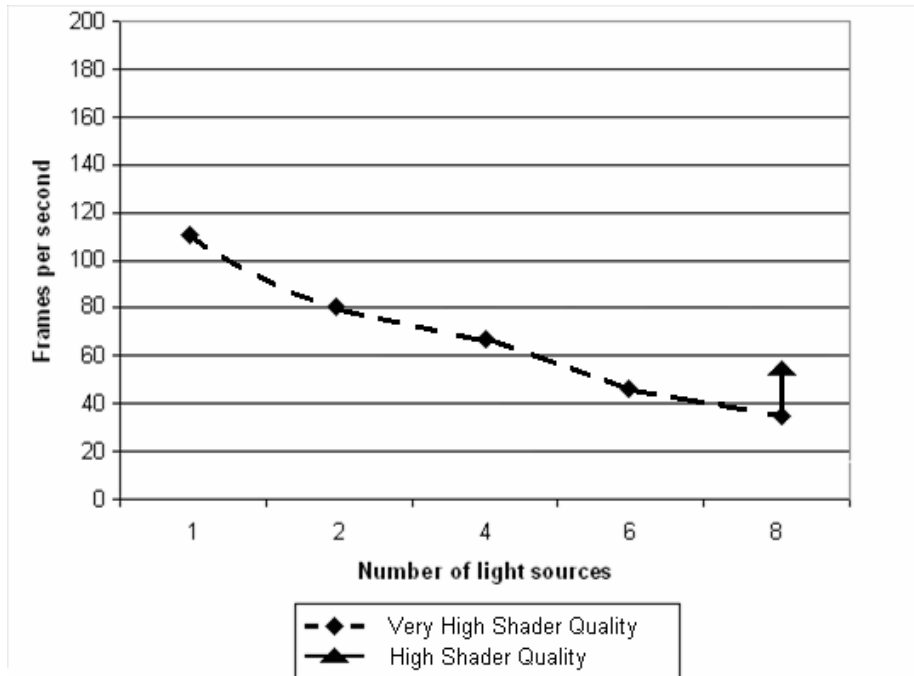


Figure 5.8 Shader performance data for up to eight light sources.

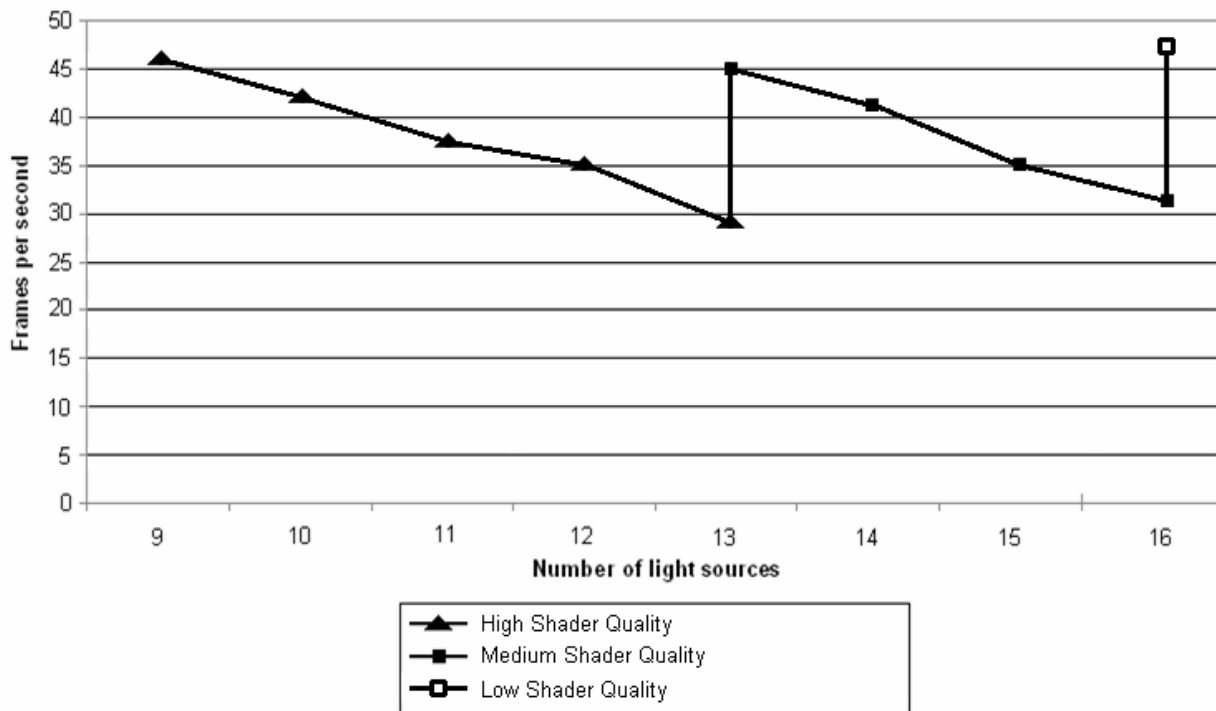


Figure 5.9 Shader performance data for nine to sixteen light sources.

Local Illumination

For local illumination quality scaling, the benchmarking environment (excluding all other special effects) consisted of five dynamic light sources. Fifty additional light sources were then progressively added (with a number of objects positioned relatively close to the viewer). This allowed the transition from the High Local Illumination Quality grouping to the Low Local Illumination Quality Grouping to be analysed.

The scene's lighting was now reset to twenty dynamic light sources – the High Local Illumination Quality grouping was successfully selected.

Figure 5.10 shows the performance data obtained (for this specific instance) for up to twenty-five light sources with Figure 5.11 showing the results obtained for thirty to sixty-five light sources.

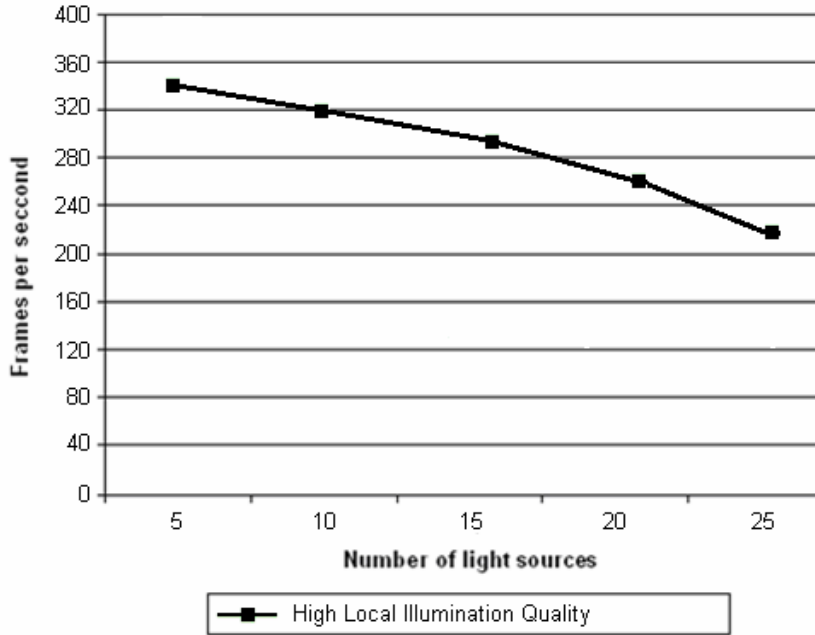


Figure 5.10 Shader performance data for up to twenty-five light sources.

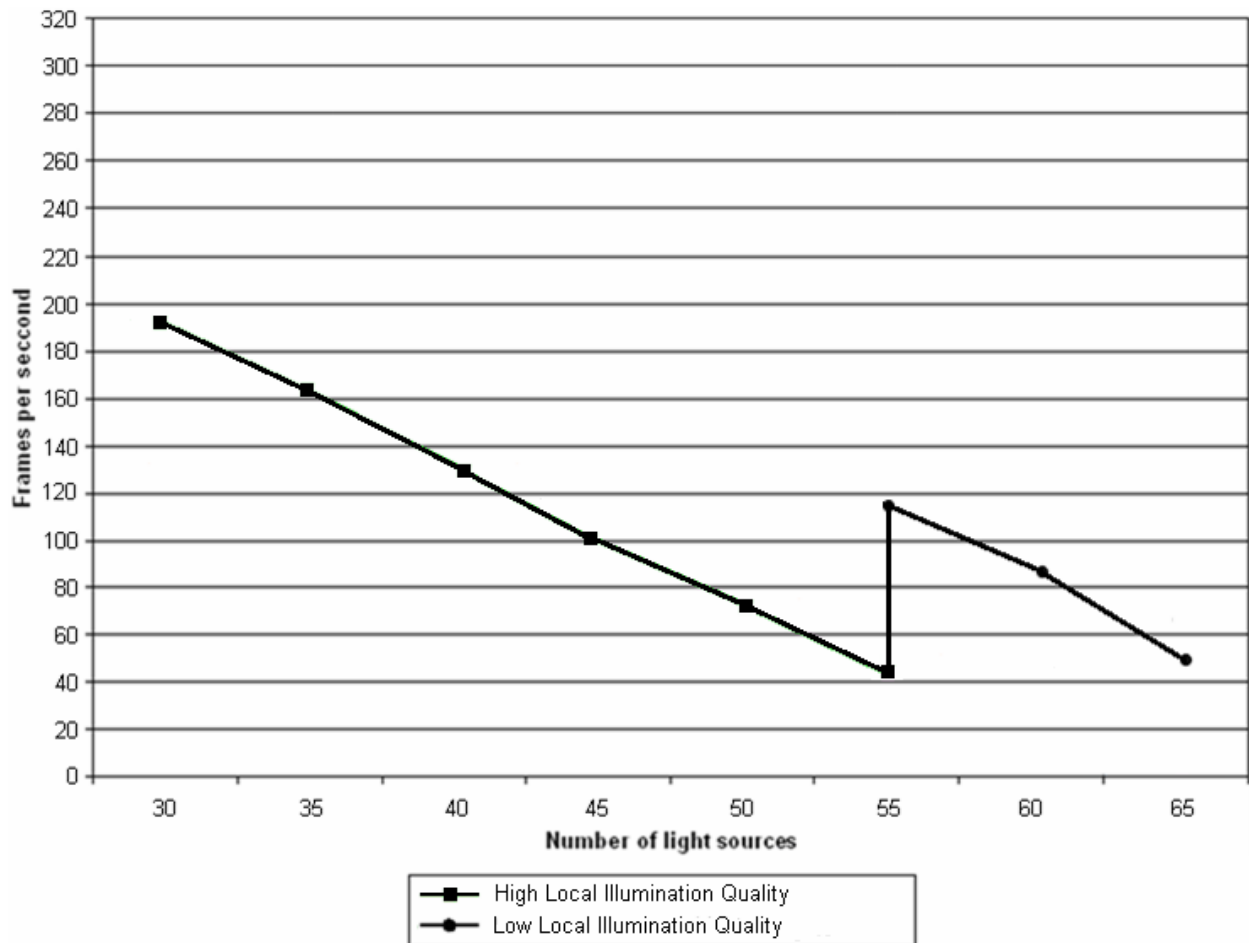


Figure 5.11 Shader performance data for thirty to sixty-five light sources.

Reflection and Refraction

Benchmarking the renderer’s reflection and refraction quality scaling mechanism commenced with a test environment consisting of a single light source. Seven additional light sources, with a number of objects positioned relatively close to the viewer, were subsequently added. This allowed us to analyse the transition from the High Reflection and Refraction Quality grouping to the Medium Reflection and Refraction Quality Grouping.

Following this we increased the number of dynamic light sources to thirteen. The scene previously rendered using the Medium Reflection and Refraction Quality grouping were now rendered using the Low Reflection Quality Grouping.

The scene’s lighting was now reset to seven dynamic light sources – the Very High Shader Quality grouping was successfully selected.

Figure 5.12 shows the performance data obtained (for this specific instance) for up to eight light sources with Figure 5.13 showing the results obtained for nine to sixteen light sources.

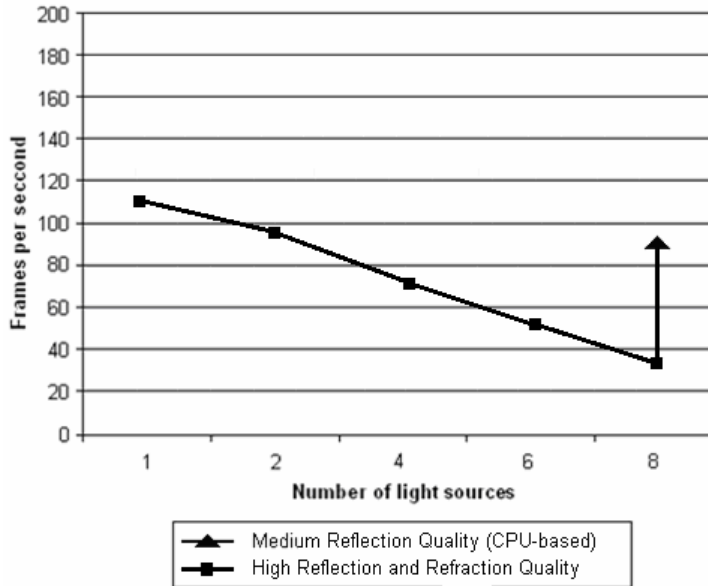


Figure 5.12 Reflection and Refraction performance data for up to eight light sources.

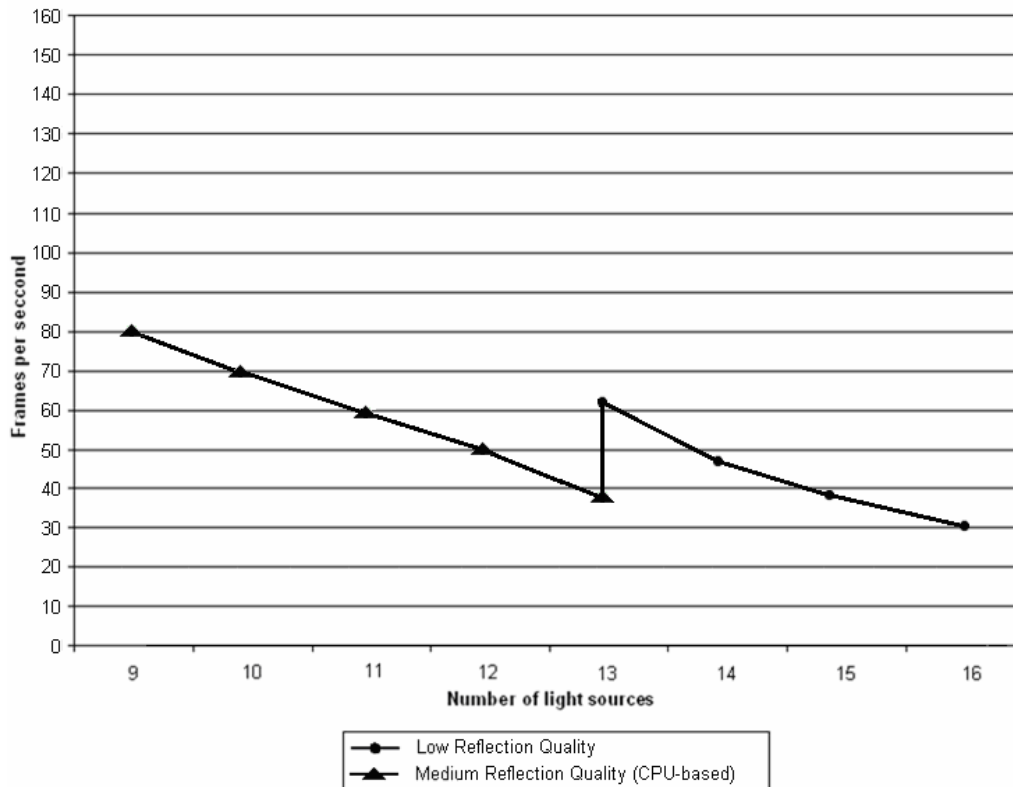


Figure 5.13 Reflection and Refraction performance data for nine to sixteen light sources.

Physics

Next, considering the renderer’s physics quality scaling, our benchmarking environment consisted of a relatively simple cubic environment featuring 3D models and a simulated environment allowing for object interaction and collision.

We started with twenty-five objects then added fifty additional interacting objects. This allowed us to analyse the transition from the Very High Physics Simulation selection to the High Physics Simulation selection (with processes efficiently distributed between the CPU and GPU).

Following this we increased the number of objects to one hundred and twenty-five. The scene previously rendered using the High Physics Simulation selection were now rendered using the Low Physics Simulation selection.

The scene’s object-count was now reset to fifty – Very High Physics Simulation was successfully selected.

Figure 5.14 shows the performance data obtained (for this specific instance) for up to one hundred and twenty-five objects.

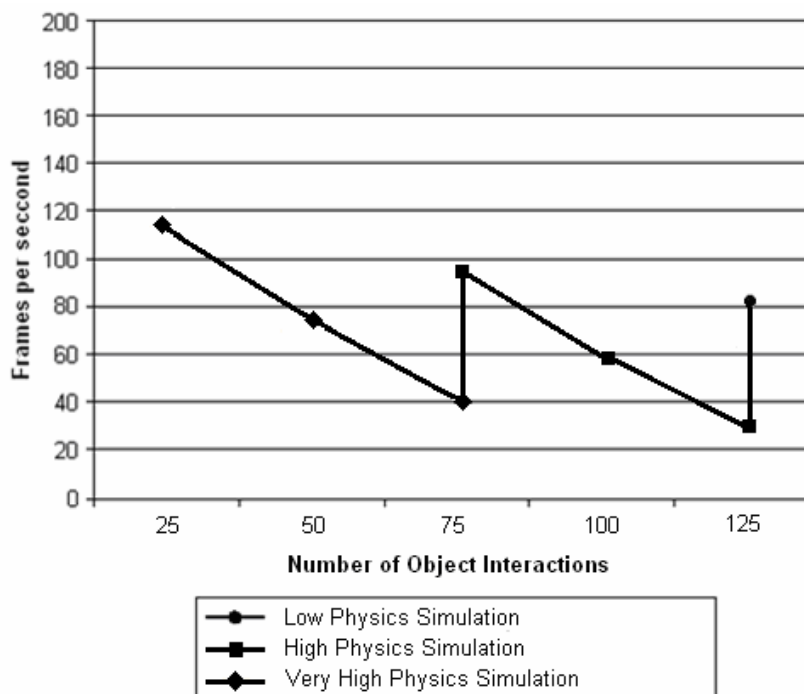


Figure 5.14 Physics performance data for up to 125 interacting objects.

Particles

As previously discussed, the particle system’s evaluation focuses on a number of particle simulations (organised into performance-impacting selections ranging from Low to Very High). To gather the necessary results, we implemented our particle system for a basic scene – a relatively simple cubic environment featuring 3D models and a simulated environment with particle effects added to simulate explosions, dust, tread marks, beams, etc (the number of particles used per simulation range from 1500 to 7500).

The experiment started with one thousand five hundred particles. Four thousand five hundred additional particles were subsequently added. This allowed us to analyse the transition from the Very High Particle Simulation selection to the High Particle Simulation selection (with physics calculations efficiently distributed between the CPU and GPU).

Following this we increased the number of particles to one nine thousand. The scene previously rendered using High Particle Simulation were now rendered using Medium Particle Simulation.

The scene’s particle-count was now reset to three thousand – Very High Particle Simulation was successfully selected.

Figure 5.15 shows the performance data obtained (for this specific instance) for up to nine thousand particles. Similar results are observed when repeating the experiment in reverse order.

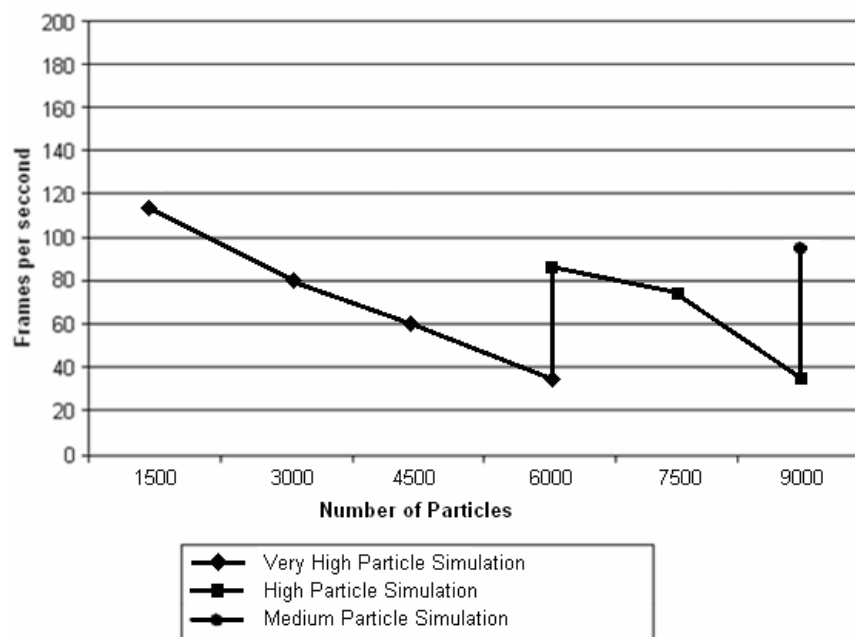


Figure 5.15 Particle performance data for up to 9000 particles.

Post-Processing

Post-Processing quality scaling benchmarking started with a basic 5 light source environment. Ten additional light sources were then added (with a number of objects positioned relatively close to the viewer). A transition from the High Post-Processing Quality grouping to the Medium Post-Processing Quality grouping was observed.

Following this we increased the number of dynamic light sources to twenty-five. The scene previously rendered using the Medium Post-Processing Quality grouping were now rendered using the Low Post-Processing Quality grouping.

The scene's lighting-count was now reset to seven dynamic light sources – the High Post-Processing Quality grouping was successfully selected.

Figure 5.16 shows the performance data obtained for up to twenty-five light sources. Similar results, as with all the other algorithms and approaches, are observed when repeating the experiment in reverse order.

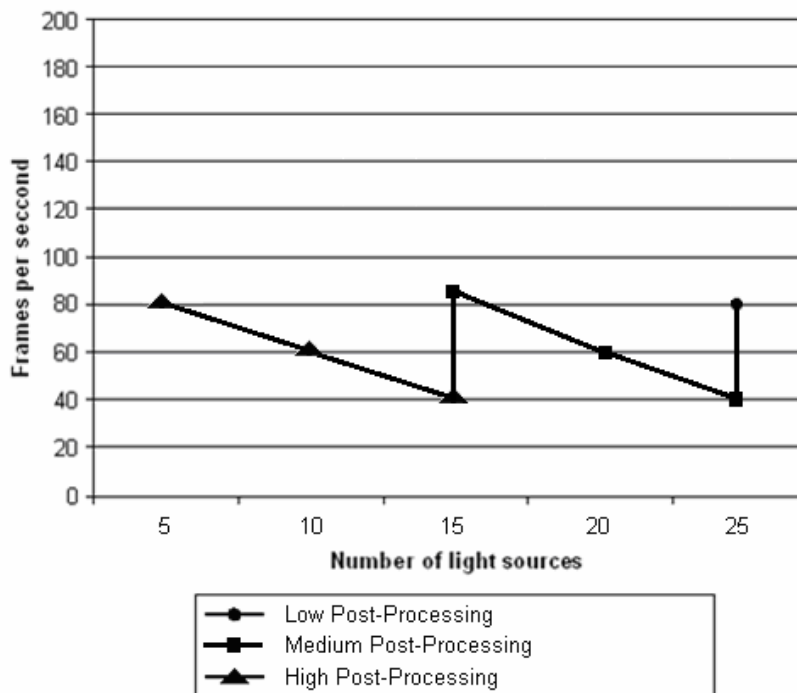


Figure 5.16 particle performance data for up to 25 light sources.

5.5 Summary

This chapter presented the general architecture of our empirically derived system for high-speed rendering – the dynamic process allocation and selection system being the

main focus. Fuzzy logic-based reasoning for the explicit symbolisation of data was also looked at.

The final section summarised the results obtained by dynamically cycling through algorithms and quality groupings to compensate for performance-impacting changes in our rendering environment. These results illustrated the performance gains to be derived by the proposed system. The next chapter gives an overall summary of our work. It closes by discussing possible future work based on the presented research.

Summary and Conclusion

Chapter 6 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 and future work

6.1 Summary

The thesis presented a study performed through the implementation of a wide and representative range of rendering and physics algorithms (organised into performance-impacting groups). A platform supporting the swapping out of rendering algorithms and physics calculations as well as the transfer of specific tasks between the CPU/GPU was built. This platform enabled the detailed benchmarking of the various implemented algorithms which, in turn, allowed for the definition of a fuzzy-logic based expert system that was embedded into a real-time rendering engine. The rendering engine analyses the 3D environment being rendered and uses the benchmarked performance data that has been encapsulated in the fuzzy-logic based selection engine to determine the best solution to a given problem at any given moment. Whenever appropriate and for cube mapping and physics calculations, it augments the computational power of the parallel compute engine in modern GPUs with that of multi-core CPUs. This allowed for the rendering of complex geometric environments through the real-time swapping of rendering algorithms and, as proof of concept, through the effective distribution of specific processing tasks between the CPU and GPU.

The thesis was divided into two parts. Part I provided the background material deemed necessary to arrive at the final result. It started by looking at game engine architecture in general, highlighting the importance of software componentry, and the difference between game-engine code and game-specific code. Following this it focussed on a number of game engine architectures, specifically ad-hoc, modular and the directed acyclic graphs architecture (DAG).

Next it considered the first step invoked whenever a game is executed, namely initialisation. Initialisation was described as the stage responsible for resource and device acquisition, memory allocation, setup of the game's GUI, loading of art assets, etc. Following front-end initialisation, it discussed the exit state and the game loop for the uninterrupted execution of a game.

Following this, the thesis dealt with the general design and implementation of a generic game engine which serves as the core of the presented dynamically scalable interactive rendering engine.

The thesis then introduced our modular rendering engine as a scalable interactive testing environment and complete solution for the rendering of computationally intensive 3D environments. A detailed discussion of the presented interactive environment's core rendering elements was subsequently given. These elements were grouped into the following rendering or computation categories: shaders, local illumination, reflection and refraction, shadows, physics, particles and post-processing special effects. This was the end of Part I.

Part II of the thesis categorised the presented algorithms and rendering groupings based on the level-of-detail/rendering quality and the associated computational impact. It also focused on the critical analysis and detailed benchmarking of the presented rendering and simulation techniques – the data used by the presented fuzzy-based selection and allocation system.

Part II commenced with a discussion of the proposed benchmarking mechanism as well as a set of criteria for the evaluation of rendering algorithms and techniques. The given evaluation criteria were selected with the aim of assessing the relationship between rendering quality and performance – in turn allowing for, where applicable, the isolation of key algorithmic weaknesses and possible bottleneck areas.

Drawn from the MSc dissertation preceding this thesis (2008), the shadow 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 Rautenbach et al's algorithm based on shadow volumes, spatial subdivision and instruction set utilisation.

Shader evaluation subsequently focused on a number of shader implementations and lighting approaches. Following this, two local illumination configurations were investigated – the first of these limiting the number of light sources in an attempt to reduce GPU utilisation with the second lifting this limitation while occluding local light sources (a technique used to approximate the effect of environment lighting as an attempt to simulate the way light radiates in real life).

Next the evaluation focused on a number of reflection and refraction implementations and approaches, specifically: basic environmental mapping, CPU-based cube mapping, refractive environmental mapping and the extension of these reflection and refraction algorithms through the addition of the Fresnel effect and chromatic dispersion.

The thesis then shifted focus to the evaluation of a number of physics calculations such as object acceleration, force, linear momentum, gravitational pull, projectile simulation through trajectory paths, friction and collision detection followed by the benchmarking of the presented rendering engine's dynamically allocated particle generator.

The benchmarking exercise concluded with the performance analysis of a number of post-processing shader implementations and lighting approaches, specifically displacement mapping, bloom effects, ambient occlusion, depth of field and halo effects.

The thesis closed by presenting the general architecture of the proposed dynamically scalable interactive rendering engine – the dynamic process allocation and selection

system being the main focus. It also looked at fuzzy logic-based reasoning for the explicit symbolisation of data. The results obtained by dynamically cycling through and offloading algorithms and quality groupings to compensate for performance-impacting changes in a rendering environment were subsequently given. These results illustrated the performance gains inherent to the proposed system's use.

6.2 Concluding Remarks and Future Work

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 rendering algorithms are only amenable to specific rendering conditions and/or situations.

A viable solution to GPU and, to a limited degree, CPU over- and/or underutilisation (depending on the scene being rendered) was to perform a critical analysis of numerous rendering algorithms with the aim of assessing the relationship between rendering quality and performance. Using this performance data gathered during the analysis of various algorithms, we were able to define a fuzzy logic-based selection engine to control the real-time selection of rendering algorithms and special effects groupings based on environmental conditions (as discussed in Chapter 4 and 5). This system ensures the following: nearby effects are always of high-quality (where computational resources are available), distant effects are, under certain conditions, rendered at a lower quality and the frames per second rendering performance is always maximised. Furthermore, as a secondary objective, we have shown that the unification of the parallel compute engine present in modern GPUs with that of multi-core CPUs to allow for the rendering of complex geometric environments is a viable solution for the management of scarce computational resources and that improved rendering quality and performance can be achieved through load-balancing between the CPU and GPU.

It is important to note that this engine and its selective utilisation of the CPU in an attempt to free up GPU resources and, in turn, to accelerate graphics performance is, in principle, also adaptable for use with 3D capable mobile devices (such as the iPhone, iPad and iPod Touch); it is expected to give these devices the ability to render special effects not previously possible by maximising the utilisation of both CPU and GPU. Further experimentation in this regard would seem appropriate.

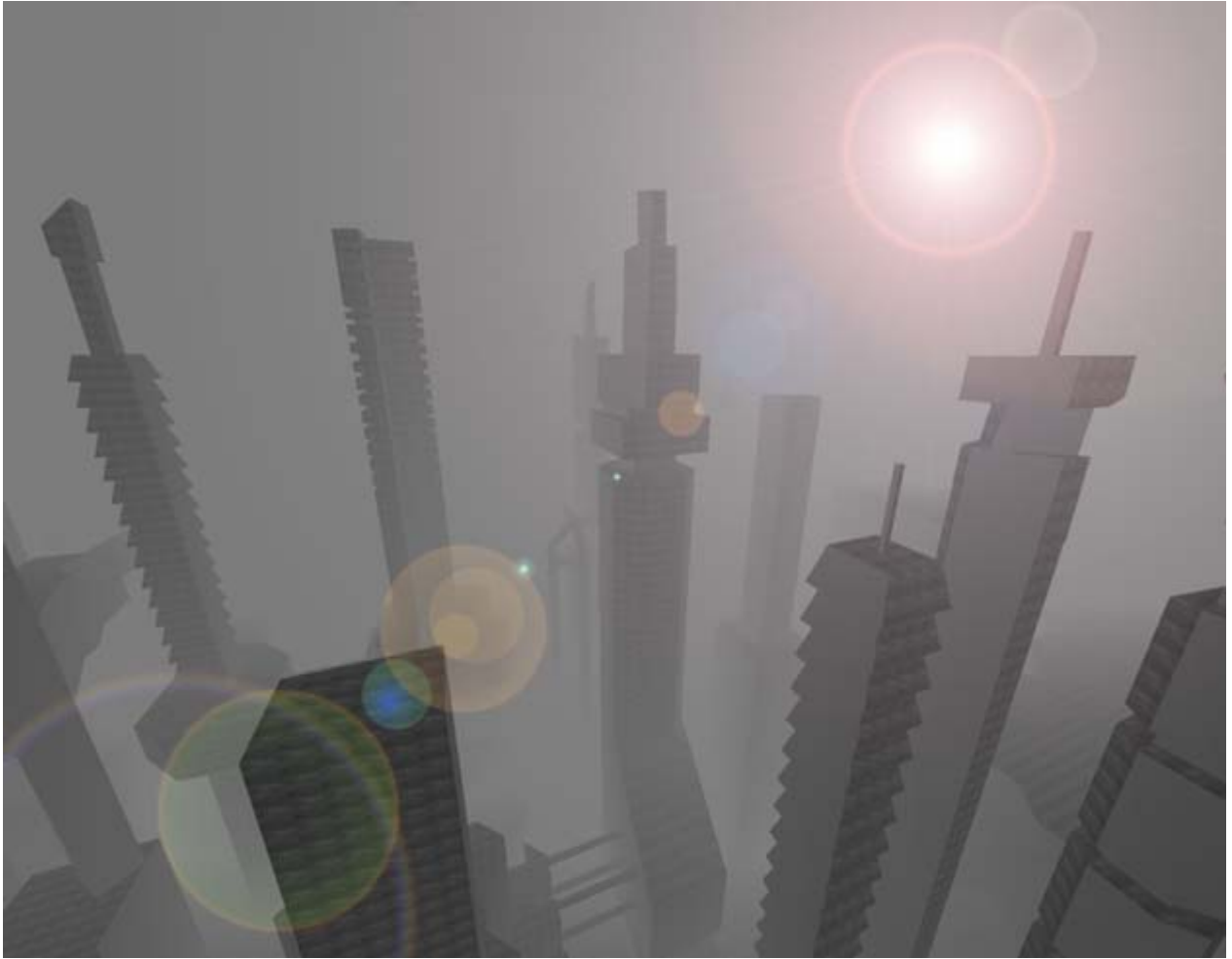
We have also demonstrated that the use of a relatively simple fuzzy-logic based expert system can serve as a viable solution to the problem of selecting between and distributing competing algorithms in real-time. This resulted in the optimisation of GPU usage by ensuring that the quality of special effects is appropriately tuned.

This work is also, in some sense, similar to current research on software evolution in the context of MAUS (Mobile and Ubiquitous Systems) which investigates how on-the-fly architectural reconfigurations are needed for such systems as context changes due to their mobility (Autili et al, 2010). Our work can inform theirs in as much as it points to the utility of a fuzzy-logic based expert system to determine which changes to make as the context changes.

It is also important to note that, despite all the rendering algorithms and approaches available, a lot of work remains in the field. More algorithms could, as future work, be benchmarked and added to our selection engine's knowledge base. Special effects groupings could also be assigned collective weights based on the groups overall impact on rendering performance (for example, the post-processing effects group will have a bigger overall performance impact than the local illumination group). The implemented rendering engine is also highly expandable and alternate rendering solutions, whether GPU or CPU based, can be implemented and loaded into the engine as additional dynamic link libraries. Alternate algorithmic performance improvements can also be pursued.

Furthermore, utilising a selection system such as the one in this thesis will allow modern engines to not only do away with their performance setup screens (thus freeing users from the cumbersome task of fine-tuning the game's graphics performance) but will guarantee a rendering environment that is running at the most optimised level possible by not just lowering "drawing distance" or "texture quality" but by actually selecting the most appropriate rendering approach and shader implementation for the current scene being rendered.

Immersive rendering approaches used in conjunction with AI subsystems, game networking and logic, physics processing and other special effects (such as post-processing shader effects) are immensely processor intensive and can only be collectively implemented on high-end hardware. This thesis has illustrated that by cycling and distributing algorithms based on environmental conditions and by the exploitation of algorithmic strengths, that a vast array of high-quality real-time special effects and highly accurate calculations can become as common as texture mapping.



References

Akeley K. and Jermoluk T. (1988). High Performance Polygon Rendering. *Computer Graphics*, 22(4).

Akeley K. (1993). Reality Engine Graphics. *Computer Graphics*.

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*.

Alard J. and Raffin B. (2005) A Shader-Based Parallel Rendering Framework. *IEEE Visualization Conference Proceedings, 2005*.

Angel E. (1990). Computer Graphics. *Addison-Wesley*.

Angel E. (2006). Interactive Computer Graphics (fourth edition). *Addison-Wesley*.

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>

Arvo J. and Kirk D. (1987) Fast Ray Tracing by Ray Classification. *Computer Graphics*, 21(4).

Arvo J. (Ed.). (1991). Graphics Gems II. *Academic Press*.

Atherton P., Weiler K. and Greenberg D. (1978) Polygon Shadow Generation. *Computer Graphics*, 12(3).

August D., Huang J., Jablin T., Kim H., Mason T., Prabhu P., Raman A. and Zhang Y (2011) Fundamentals of Multi-core Software Development. ISBN: 978-1439812730. *Chapman & Hall / CRC Press, December 2011*.

Autili M., Inverardi P., Tivoli M. (2010). Assessing dependability for mobile and ubiquitous systems: Is there a role for Software Architectures? *Quality Software (QSIC), 2010 10th International Conference*.

Banchoff T. and Werner J. Linear Algebra through Geometry. *Springer-Verlag*.

Bier E. and Sloan K. (1986). Thow-part Texture Mapping. *IEEE Computer Graphics and Applications*, 6(9).

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

Belleman R., Bedorf J., Zwart S. (2008) High Performance Direct Gravitational N-body Simulations on Graphics Processing Units II: An Implementation in CUDA. *New Astronomy*, vol. 13, no. 2, 2008, pp. 103-112.

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

Bernhardt A., Maximo A., Velho L., Hnaidi H. and Cani M. (2011) Real-time Terrain Modeling using CPU-GPU Coupled Computation. *International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2011*.

Blinn J. (1977). Models of Light Reflection for Computer Synthesized Pictures. *Computer Graphics*, 11(2).

Blinn J. (1978). Simulation of Wrinkled Surfaces. *Computer Graphics*, 12(3).

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

Blinn J. and Newell M. (1976). Texture and Reflection in Computer Generated Images. *Communications of the ACM*, 19(10).

Blinn J. (1977). Models of Light Reflection for Computer-Synthesized Pictures. *Computer Graphics*, 11(2).

Blizzard. (2010) StarCraft II Effects & Techniques. Details published online at: <http://developer.amd.com/documentation/presentations/legacy/Chapter05-Filion-StarCraftII.pdf>

Borges C. (1991). Trichromatic Approximations for Computer Graphics Illumination Models. *Computer Graphics*, 25(4).

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*.

Bowyer A. and Woodwark J (1983). A Programmer's Geometry. *Butterworth*.

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

Bresenham J. (1987). Ambiguities in Incremental Line Rastering. *IEEE Computer Graphics and Applications*.

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.

Cabral B., Max N. and Springmeyer R. (1987). Bidirectional Reflection Functions from Surface Bump Maps. *Computer Graphics*, 21(4).

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/>

Carlbon I. and Paciorek J. (1978). Planar Geometric Projection and Viewing Transformations. *Computing Surveys*, 10(4).

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

Carmack J. (2011) Doom Creator More Excited About Games Now Than Ever. Published online at: <http://www.industrygamers.com/news/doom-creator-more-excited-about-games-now-than-ever//article.php?aid=532>.

Castleman, K. (1996). *Digital Image Processing*. Prentice Hall.

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

Cook R. and Torrance K. (1982). A Reflectance Model for Computer Graphics. *Computer Graphics*, 15(3).

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

Cohen J., Olano M. and Manocha D. (1998) Appearance-Preserving Simplification. *SIGGRAPH 1998*.

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

Crockett T. (1995) Parallel Rendering. *Institute for Computer Applications in Science and Engineering, NASA Langley Research Center*.

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

Crow F. (1981). A Comparison of Antialiasing Techniques. *IEEE Computer Graphics and Applications*, 1(1).

Crossno P. and Angel E. (1997). Isosurface Extraction Using Particle Systems. *IEEE Visualization*.

Crytek (2011) Crytek 2 Key Rendering Features. Published online at: <http://crytek.com/sites/default/files/Crysis%20%20Key%20Rendering%20Features.pdf>

DeLoura M (Ed.). (2000). *Game Programming Gems*. Charles River Media.

DeRose T. (1988). A Coordinate Free Approach to Geometric Programming. *1988 SIGGRAPH Course Notes*.

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.

Duchaineua M., Wolinsky M. and Sigeti D. (1997). ROAMing Terrain: Real-time Optimally Adapting Meshes. *IEEE Proceedings of Visualization 1997*.

Eberly, D. (2001). 3D Game Engine Design. *Morgan Kaufman*.

Ebert D., Musgrave D., Peachey D., Perlin K. and Worley S. (2002). Texturing and Modeling, A Procedural Approach (third edition). *Morgan Kaufman*.

Enderle G., Kansy K., Pfaff G. (1984). Computer Graphics Programming: GKS – The Graphics Standard. *Springer-Verlag*.

Epic Games (2012). Unreal Technology Features. Published online at: <http://www.unrealengine.com/features/rendering>.

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.

Fangerau J., Krömker S. (2010) Parallel Volume Rendering Implementation on Graphics Cards using CUDA. *LNCS 6310, pp. 143-153. Springer-Verlag. Berlin*.

Farin G. (1988). Curves and Surfaces for Computer Aided Geometric Design. *Academic Press*.

Fernando R. and Kilgard M. (2003). *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley.

Fernando R. (2004). *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley.

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

Flynt J. and Salem O. (2004). *Software Engineering for Game Developers*. Course Technology PTR.

Foley J., van Dam A., Feiner S. and Hughes J. (1990). *Computer Graphics (second edition)*. Addison-Wesley.

Fosner R. (1996). *OpenGL Programming for Windows 95 and Windows NT*. Addison-Wesley.

Fuchs H., Kedem Z. and Naylor B. (1980). On Visible Surface Generation by A Priori Tree Structures. *SIGGRAPH 80*.

Funge J. (1999). *AI for Computer Games and Animation: A Cognitive Modeling Approach*. AK Peters.

Future Chips (2011) Tutorial on Removing Branches. Published online at: <http://www.futurechips.org/tips-for-power-coders/basic-technique-to-help-branch-prediction.html>

Gallagar R. (1995). *Computer Visualization: Graphics Techniques for Scientific and Engineering Analysis*. CRC Press.

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

Glassner A. (Ed.). (1990). *Graphics Gems I*. Academic Press.

Greene N. (1986). Environment Mapping and Other Applications of World Projections. *IEEE Computer Graphics and Applications*.

Goral C, Torrance D., Greenberg D. and Battaile B. (1984). Modeling the Interaction of Light Between Diffuse Surfaces. *Computer Graphics*, 18(3).

Gouraud H. (1971). Computer Display of Curved Surfaces. *IEEE Transactions on Computers*.

Gray K. (2003). The Microsoft DirectX 9 Programmable Graphics Pipeline. *Microsoft Press*.

Hahn J. (1988). Realistic animation of rigid bodies. *Communications of the ACM*.

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

Hall R. (1989). Illumination and Color in Computer Generated Imagery. *Springer-Verlag*.

Halliday D., Resnick R. and Walker J. (2007). Fundamentals of Physics Extended. *Wiley*.

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

Hasenfratz J., Lapierre M., Holzschuch N. and Sillion F. (2003). *Computer Graphics Forum*, 22(4):753–774.

Hearn D. and Baker M. (2004). Computer Graphics (third edition). *Prentice Hall*.

Heckbert P. and Hanrahan P. (1984). Beam tracing Polygonal Objects. *Computer Graphics*, 18(3).

Heckbert P. (1986). Survey of Texture Mapping. *IEEE Computer Graphics and Applications*, 6(11).

- Heckbert P. (Ed.). (1994). Graphics Gems IV. *Academic Press*.
- Hecker C. (2000). Physics in computer games. *Communications of the ACM*.
- Heidmann T. (1991) Real shadows real time. *IRIS Universe*, 18:28–31.
- Hill F. (2001). Computer Graphics (second edition). *Prentice Hall*.
- 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.
- Hermann E., Raffin B., Faure F., Gautier T., Allard J. (2011) Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*.
- Hopgood F., Duce D., Gallop A. and Sutcliffe D. (1983). Introduction to the Graphical Kernel System: GKS. *Academic Press*.
- Hoppe H. (1996). Progressive Meshes. *1996 SIGGRAPH Proceedings*.
- Hoppe H. (1998). Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. *IEEE Proceedings of Visualization*.
- Hourcade J.-C. and Nicolas A. (1985) Algorithms for antialiased cast shadows. *Computers & Graphics*, 9(3):259–265.
- Huang J., Raman A., Zhang Y., Jablin T., Hung T., and August D. (2010) Decoupled Software Pipelining Creates Parallelization Opportunities. *Proceedings of the 2010 International Symposium on Code Generation and Optimization (CGO), April 2010*.
- Hubbard P. (1996). Interactive Collision Detection. *IEEE Transactions on Visualization and Computer Graphics*.
- Id Software. (2011) Rage and the Tech Behind id Tech 5. Details published online at: <http://hothardware.com/Reviews/Rage-The-Tech-Behind-Id-Tech-5/>
- 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/asmona/eng/popular/20240.htm

Intel. (2010) Programmatically Tracking CPU Utilization. Published online at: <http://software.intel.com/en-us/articles/programmatically-tracking-cpu-utilization/>.

Intel. (2011) Hybrid Rendering. Published online at: <http://www.intel-vci.uni-saarland.de/en/projects/hybrid-rendering.html/>.

Isard M., Shand M., and Heirich A. (2002) Distributed rendering of interactive soft shadows. *Fourth Eurographics Workshop on Parallel Graphics and Visualization*.

Jablin T., Prabhu P., Jablin J., Johnson N., Beard S., August D. (2011) Automatic CPU-GPU Communication Management and Optimization. *PLDI'11 (Programming Language Design and Implementation), June 4–8, 2011, San Jose*.

Jablin T., Jablin J., Prabhu P., Liu F., and August D. (2012) Dynamically Managed Data for CPU-GPU Architectures. *Proceedings of the 2012 International Symposium on Code Generation and Optimization (CGO), March 2012*.

Joselli M., Clua E., Montenegro A., Conci A., Pagliosa P. (2008) A New Physics Engine with Automatic Process Distribution between CPU-GPU. *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*.

Joselli M., Zamith M., Clua E., Montenegro A., Leal-Toledo R., Conci A., Pagliosa P., Valente L., Feijó B. (2009) An Adaptive Game Loop Architecture with Automatic Distribution of Tasks between CPU and GPU. *Computers in Entertainment (CIE), Volume 7 Issue 4*.

Kalogirou, H. (2006) How to do Good Bloom for HDR Rendering. Published online at: <http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/>

Kajiya J. (1986). The Rendering Equation. *1986 SIGGRAPH Proceedings*.

Kam T., Villa T., Brayton R. and Sangiovanni-Vincentelli A. (1996). Synthesis of Finite State Machines: Functional Optimization. *Kluwer Academic Publishers*.

Kerr A., Damos G., Yalamanchili S. (2010) Modeling GPU-CPU Workloads and Systems. *GPGPU '10 Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*.

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.

Kilgard M. (1994). An OpenGL Toolkit. *The X Journal*.

Kilgard M. (1994). OpenGL and X, Part 3: Integrated OpenGL with Motif. *The X Journal*.

Kilgard M. (1996). OpenGL Programming for the X Windows System. *Addison-Wesley*.

Kilgard M. (1996). The OpenGL Utility Toolkit (GLUT) Programming Interface. *Silicon Graphics*.

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

Kirk D. (Ed.). (1992). Graphics Gems III. *Academic Press*.

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*.

Lane J. and Riesenfeld R. (1980). A Theoretical Development for the Computer Generation and Display of Piecewise Polynomial Surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1).

Lane J., Carpenter L., Whitted T. and Blinn J. (1980). Scan Line Methods for Displaying Parametrically Defined Surfaces. *Communications of the ACM*, 23(1).

Langer and Bülthoff (2000) Depth discrimination from shading under diffuse lighting. *Perception* 29(6) 649 – 660.

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

Lay D. (2005). Linear Algebra and Its Applications (third edition). *Addison-Wesley*.

Legakis J. (1998). Fast Multi-Layer Fog. *ACM SIGGRAPH 98 Conference abstracts and applications*.

Lengyel E. (2003). Mathematics for 3D Game Programming and Computer Graphics (second edition). *Charles River Media*.

Leon. S. (2006). Linear Algebra with Applications. *Prentice Hall*.

Levoy M. (1988). Display of Surface from Volume Data. *IEEE Computer Graphics and Applications*, 8(3).

Levoy M. and Hanrahan P. (1996). Light Field Rendering. *1996 SIGGRAPH Proceedings*.

Liang Y. and Barsky B. (1984). A New Concept and Method for Line Clipping. *ACM Transactions on Graphics*, 3(1).

Lindstrom P. and Pascucci V. (2001). Visualization of Large Terrains Made Easy. *IEEE Proceedings of Visualization 2001*.

Linholt E., Kilgard M. and Morelton H. (2001). A User-Programmable Vertex Engine. *SIGGRAPH 2001*.

Lindholm et al. (2008) NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, vol. 28, no. 2, 2008, pp. 39-55.

Likun Z. and Dingfang C. Accelerate Your Graphic Program with GPU/CPU Cache. *Proceedings of the 2008 International Conference on Cyberworlds*.

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

Maxwell E. (1951). General Homogeneous Coordinates in Space of Three Dimensions. *Cambridge University Press*.

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 included CD).

Microsoft. DirectX 10 SDK Documentation.

Microsoft. Platform SDK C++ Documentation.

Mikkelsen M. (2008) Simulation of Wrinkled Surfaces Revisited. *Master's thesis, Department of Computer Science at the University of Copenhagen, Pages: 1-109*

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

Möller T. and Haines E. (2002). Real-Time Rendering (second edition). *A K Peters*.

Montrym J., Baum D., Dignam D. and Migdal C. (1997). InfiniteReality: A Real-Time Graphics System. *Computer Graphics*.

Moore M. and Wilhelms J. (1988). Collision Detection and Response for Computer Animation. *Computer Graphics*, 22(4).

Moore S. (2011) With Denver Project NVIDIA and ARM Join CPU-GPU Integration Race. Published online at: <http://spectrum.ieee.org/tech-talk/semiconductors/processors/with-denver-project-nvidia-and-arm-join-cpugpu-integration-race>

Murray J. and van Ryper W. (1994). Encyclopaedia of Graphics File Formats. *O'Reilly and Associates*.

Newman W. and Sproull R. (1973). Principles of Interactive Computer Graphics. *McGraw-Hill*.

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

Nickolls J., Dally W. (2010) The GPU Computing Era. *IEEE Micro*, vol. 30, no. 2, pp. 56-69.

Nickolls J., Kirk D. (2009) Graphics and Computing GPUs. *Computer Organization and Design: The Hardware/Software Interface, D.A. Patterson, and J.L. Hennessy 4th ed., Morgan Kaufmann, 2009, pp. A2-A77.*

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

Nishita T. and Nakamae E. (1985). Continuous Tone Representation of 3D Objects taking account of Shadows and Interreflection. *Computer Graphics*, 19(3).

Novins K., Sillion F. and Greenberg D. (1990). An Efficient Method for Volume Rendering using Perspective Projections. *Computer Graphics*, 24(5).

NVIDIA. (2009) Fermi: NVIDIA's Next Generation CUDA Compute Architecture. Published online at: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

NVIDIA. (2011) NVIDIA PerfKit: Performance Tools to Help Debug and Profile OpenGL and Direct3D applications. Published online at: <http://developer.nvidia.com/nvidia-perfkit>.

NVIDIA. (2011) Advanced parallax mapping techniques. Published online at: <http://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html>.

Ohshima S., Kise K., Katagiri T., Yuba T. (2006) Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment. *Proceedings of the 7th international conference on High performance computing for computational science.*

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.*

Paeth A. (Ed.). (1995). Graphics Gems V. *Academic Press.*

Pajot A., Barthe L., Paulin M. and Poulin P. (2011) Combinatorial Bidirectional Path-Tracing for Efficient Hybrid CPU/GPU Rendering. *Eurographics '11*.

Pavlidis T. (1995). Interactive Computer Graphics in X. *PWS Publishing*.

Peachey D. (1985). Solid Texturing of Complex Surfaces. *Computer Graphics*, 19(3).

Percy M., Airey J. and Cabral B. (1997). Efficient Bump Mapping Hardware. *1997 SIGGRAPH Proceedings*.

Perumalla K., Aaby B. (2008) Data Parallel Execution Challenges and Runtime Performance of Agent Simulations on GPUs. *Proceedings of the 2008 Spring simulation multiconference*.

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).

Piegl L. (1993) Fundamental Developments of Computer-Aided Geometric Modelling. *Academic Press*.

Policarpo F. and Oliveira M. (2006) Relief Mapping of Non-Height-Field Surface Details. *ACM SIGGRAPH 2006 Symposium on Interactive 3D Graphics*.

Porter T. and Duff T. (1984). Compositing Digital Images. *Communications of the ACM*.

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

Qi Ren, D. (2011) Algorithm Level Power Efficiency Optimization for CPU-GPU Processing Element in Data Intensive SIMD/SPMD Computing. *Journal of Parallel and Distributed Computing* , Volume 71 Issue 2.

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, London.

Rautenbach, P. (2008) An empirically derived system for high-speed shadow rendering. Master's thesis, Department of Computer Science, University of Pretoria, South Africa.

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)*.

Rector B. and Newcomer J. (1997). Win32 Programming. Addison-Wesley.

Reeves W. (1983). Particle Systems – a Technique for Modelling a class of Fuzzy Objects. *Computer Graphics*, 17(3).

Reeves W. and Blau R. (1985). Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems. *Computer Graphics*, 19(3).

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>

Reynolds C. (1987). Flocks, Herds, and Schools: A Distributed Behavioural Model. *Computer Graphics*, 21(4).

Riesenfeld R. (1987). Homogeneous Coordinates and Projective Planes in Computer Graphics. *IEEE Computer Graphics and Applications*, 1(1).

Rogers D. (1985). *Procedural Elements for Computer Graphics* (second edition). *McGraw-Hill*.

Rogers D. and Adams J. (1990). *Mathematical Elements for Computer Graphics*. *McGraw-Hill*.

Rogerson D. (1997). *Inside Com*. Microsoft Press.

Rossignac J. and Requicha A. (1986). Depth Buffering Display Techniques for Constructive Solid Geometry. *IEEE Computer Graphics and Applications*, 6(9).

Royce W. (1970). Managing the Development of Large Software Systems. *IEEE WESCON 26*.

Rubin S. and Whitted T. (1980). A 3D Representation for Fast Rendering of Complex Schemes. *Computer Graphics*, 14.

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.

Schaufler G. and Sturzlinger W. (1996). A 3D Image Cache for Virtual Reality. *Proceedings of the 1996 Eurographics*.

Schlick C. (1993). A Customizable Reflectance Model for Everyday Rendering. *Fourth Eurographics Workshop on Rendering*.

Schumaker R., Brand B., Guilliland M. and Sharp W. (1969). Applying Computer Generated Images to Visual Simulation. *US Airforce Human Resources Lab Technical Report: AFHRL-Tr-69*.

Seitz S. and Dyer C. (1996). View Morphing. *1996 SIGGRAPH Proceedings*.

Shade J., Gortler S., He L. and Szeliski R. (1998). Layered Depth Images. *1998 SIGGRAPH Proceedings*.

Shade J., Lischinski D., Salesin D., DeRose T. and Snyder J. (1996). Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. *1996 SIGGRAPH Proceedings*.

Shainer G., Lui P., Liu T. (2011) The Development of Mellanox/NVIDIA GPUDirect over InfiniBand: A New Model for GPU to GPU Communications. *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*.

Shirley P. (2002). Fundamentals of Computer Graphics. *AK Peters*.

Sillion F. and Puech C. (1989). A General Two-Pass Method Integrating Specular and Diffuse Reflection. *Computer Graphics*, 22(3).

Smith A. (1978). Color Gamut Transformation Pairs. *Computer Graphics*, 12.

Snyder J. (1992). Generative Modelling for Computer Graphics. *Academic Press*.

Snyder J. (1998). Visibility Sorting and Compositing without Splitting for Image Layer Decomposition. *1998 SIGGRAPH Proceedings*.

Stallings W. (2000). Operating Systems: Internals and Design Principles (fourth edition). *Prentice Hall*.

Stallings W. (2002). Computer Organization and Architecture (sixth edition). *Prentice Hall*.

Stam J. and Loop C. (2003). Quad/Triangle Subdivision. *Computer Graphics Forum* 22.

Stang G. (1993). Introduction to Linear Algebra. *Wellesley-Cambridge Press*.

Sutherland I. (1963). Sketchpad, A Man-Machine Graphical Communication System. *Proceedings of the SHARE Design Automation Workshop DAC '64*.

Sutherland I. and Hodgman G. (1974). Reentrant Polygon Clipping. *Communications of the ACM*, 17(1).

Sutherland I., Sproull R. and Schumacher R. (1974). A Characterization of Ten Hidden-Surface Algorithms. *Computer Surveys*, 6(1).

Swanson R. and Thayer L. (1986). A Fast Shaded-Polygon Renderer. *Computer Graphics*, 20(4).

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.

Troelsen A. (2000). Developer's Workshop to COM and ATL 3.0. *Wordware Publishing*.

Torrance K. and Sparrow E. (1967) Theory for Off-Specular Reflection from Roughened Surfaces, *Journal of the Optical Society of America*, 57(9).

Upstill S. (1989). The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics. *Addison-Wesley*.

Venkatasubramanian S., Vudac R. (2009) Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems. *Proceedings of the 23rd international conference on Supercomputing*.

Wagner F., Schmuki R., Wagner T. and Wolstenholme P. (2006). Modeling Software with Finite State Machines: A Practical Approach. *Auerbach*.

Walter B., Hubbard P., Shirley P. and Greenberg D. (1997). Global Illumination using Local Linear Density Estimation. *ACM Transactions on Graphics*, 16(3).

Wang L., Huang Y., Chen X., Zhang C. (2008) Task Scheduling of Parallel Processing in CPU-GPU Collaborative Environment. *Proceedings of the 2008 International Conference on Computer Science and Information Technology*.

Warn D. (1983). Lighting Controls for Synthetic Images. *Computer Graphics*, 17(3).

Warren J. and Weimer H. (2003). Subdivision Methods for Geometric Design. *Morgan Kaufmann*.

Warren J. and Schaefer S. (2004). A Factored Approach to Subdivision Surfaces. *IEEE Computer Graphics and Applications*, 24(3).

Warnock J. (1969). A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures. *University of Utah Computer Science Department Technical Report: 4-15, NTIS AD-753 671*.

Watt A. and Watt M. (1992). *Advanced Animation and Rendering Techniques. Addison-Wesley*.

Watt A. (2000). *3D Computer Graphics (third edition). Addison-Wesley*.

Weiler K. and Atherton P. (1977). Hidden Surface Removal using Polygonal Area Sorting. *Computer Graphics*, 11(2).

Wenzel C. (2006). Advanced real-time rendering in 3D graphics and games: Real-time atmospheric effects in games. *SIGGRAPH 2006*.

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).

Williams L. and Chen S. (1993). View Interpolation for Image Synthesis. *1993 SIGGRAPH Proceedings*.

Wilson G., Banzhaf W. (2009) Deployment of CPU and GPU-based Genetic Programming on Heterogeneous Devices. *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*

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

Witkin A. and Heckbert P. (1994). Using Particles to Sample and Control Implicit Surfaces. *Computer Graphics*, 28(3).

Witkin A. (Ed.). (1994). An Introduction to Physically Based Modeling. *SIGGRAPH 94*.

Wloka M. (2002). Fresnel Reflection. *NVIDIA*.

Wolfram S. (1991). Mathematica. *Addison-Wesley*.

Woo A., Poulin P. and Fournier A. (1990) A Survey of Shadow Algorithms. *Computer Graphics and Applications, IEEE, 10(6):13–32*.

Wyszecki G. and Stiles W. (1982). Color Science. *Wiley*.

Yamagiwa S., Wada K. (2009) Performance Study of Interference on GPU and CPU Resources with Multiple Applications. *IEEE International Symposium on Parallel & Distributed Processing, 1-8*.

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.

Zidan M., Bonny T., Salama K. (2011) High Performance Technique for Database Applications using a Hybrid GPU/CPU Platform. *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI*.

Zink B. (2008) Architectural Requirements for a Hybrid GPU/CPU Middleware. *Proceedings of the 15th ACM Mardi Gras conference*.

Fundamentals of the Graphics Pipeline Architecture

A pipeline is a series of data processing units arranged in a chain like manner with the output of the one unit read as the input of the next. Figure A.1 shows the basic layout of a pipeline.



Figure A.1 Logical representation of a pipeline.

The throughput (data transferred over a period of time) many any data processing operations, graphical or otherwise, can be increased through the use of a pipeline. However, as the physical length of the pipeline increases, so does the overall latency (waiting time) of the system. That being said, pipelines are ideal for performing identical operations on multiple sets of data as is often the case with computer graphics.

The graphics pipeline, also sometimes referred to as the rendering pipeline, implements the processing stages of the rendering process (Kajiya, 1986). These stages include vertex processing, clipping, rasterization and fragment processing. The purpose of the graphics pipeline is to process a scene consisting of objects, light sources and a camera, converting it to a two-dimensional image (pixel elements) via these four rendering stages. The output of the graphics pipeline is the final image displayed on the monitor or screen. The four rendering stages are illustrated in Figure A.2 and discussed in detail below.

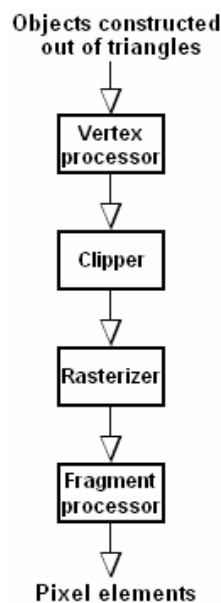


Figure A.2 A general graphics pipeline.

Summarised we can describe the graphics pipeline as an overall process responsible for transforming some object representation from local coordinate space, to world space, view space, screen space and finally display space. These various coordinate spaces are fully discussed in various introductory graphics programming texts and, for the purpose of this discussion, it is sufficient to consider the *local coordinate space* as the definition used to describe the objects of a scene as specified in our program's source code. The *world space* can be described as a coordinate space where we have a reference to the viewer's position with lighting added to our scene. *View space* is where our scene's objects are culled and clipped to determine whether an object is visible based on the position of the viewer or camera. *Screen space* is where hidden surface removal, shading and rasterization occur and it is the final stage before we enter the *display space* where the produced pixel elements are displayed via some output device (Sutherland et al, 1974). We will now look at the various stages of the graphics pipeline in detail.

A.1 Vertex Processing

The first processing unit of the graphics pipeline is the vertex processor. This processor is responsible for performing all geometric transformations and the computation of colour values of every vertex or point making up an object.

Geometric transformations (such as translations and rotations) simply refer to the process of converting the current spatial representation of an object to a different coordinate system. For example, a geometric transformation is required to represent an object, originally defined in terms of world coordinates (coordinates specified by the programmer for object representation), in terms of display coordinates (the coordinate system used by the graphics display). Each geometric transformation is defined using a matrix with a series of transformations specified by concatenating each of these matrices into a single one. Combining one matrix with another yields a third matrix that is once again combined with some other transformation matrix – an operation that clearly benefits from the use of a pipeline.

Three transformations are performed during the vertex processing stage. The first of these, namely the *modelling transformation*, takes the geometric specification of three-dimensional world objects as input. Every object, originally defined in local coordinate space, is subsequently transformed to use world-space coordinates. Each object's independent local coordinate system has now been transformed into a global coordinate system. This provides all the objects with a shared global coordinate space – i.e. one object's position can be described in terms of another's and these user defined objects can now be positioned within the same scene. All translations and rotations are performed during this transformation step.

The next transformation step, called the *viewing transformation*, transforms all world-space coordinates to coordinates specified in terms of a viewer's position and viewing direction. This transformation step leads to a viewer or camera that can be moved and rotated to any position within the world coordinate space. The original three-dimensional scene is displayed from this viewer's perspective (or point of view). Both culling (back-face elimination) and clipping are carried out in view space.

The final transformation, called the *projection transformation*, transforms the view space coordinates to two-dimensional image space or screen space so that the three-dimensional scene can be displayed on a flat plane.

The final step of the vertex processor is to assign colours, per-vertex lighting and shading to each of the vertices making up the scene (Swanson and Thayer, 1986). The rasterization stage discussed below interpolates these per-vertex lighting values for the creation of smoothly shaded lighting ranges between vertices.

A.2 Clipping and Culling

Clipping controls the field of view, i.e. managing the percentage of the world visible based on the camera's viewing angle and position. The lack of clipping does not hinder the image formation process, it is, however, crucial to ensure that this process is performed in a timely manner due to it eliminating the rendering of any unnecessary primitives that would not be visible to the viewer or camera. We define a volume similar to a stencil to block out objects not visible to the viewer. All objects and portions of objects falling outside this stencil or volume do not appear in the final image.

Clipping, unlike vertex processing, should be done on a primitive-by-primitive rather than a vertex-by-vertex basis. To accomplish this, sets of vertices are assembled into primitives, such as polygons and lines based on the implementation of some clipping algorithm such as the Cohen-Sutherland or Liang-Barsky line clipping algorithms or the Sutherland-Hodgman polygon clipping algorithm. An example illustrating the importance of clipping would be to consider a scene from a computer game consisting of numerous buildings, cars, pedestrians, shops, etc. Each of these elements are physical models stored in memory, requiring a lot of processing time for shading, texturing, animation, etc. If the scene's viewer or camera has a viewing angle of 110 degrees, then we needn't render any of the models or meshes located outside this viewing area – thus saving a lot of rendering time in the process.

Culling, or back-face elimination, refers to the process where polygons or surfaces pointing away from the camera or viewer are not rendered. For example; when a building is viewed directly from the front, then the three sides hidden from the viewer are

not drawn (shown in Figure A.3). This process, just like clipping, improves the rendering speed of a scene by reducing the number of polygons or surfaces that needs to be rendered without affecting the visual output.

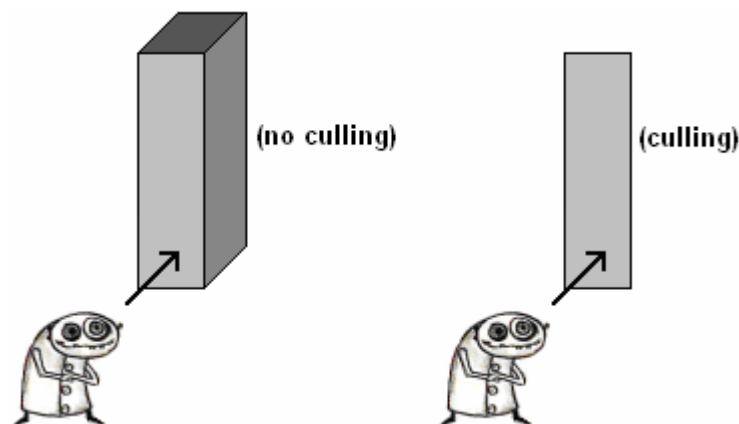


Figure A.3 Back-face elimination.

A.3 Rasterization and Fragment Processing

The rasterization, or scan conversion process converts the primitives produced by the clipper (consisting of vertices) to pixels for representation in the frame buffer and for subsequent output to a monitor. For example, a solid rectangle consisting of four vertices are transformed to two-dimensional pixels or points in the frame buffer, with these two-dimensional pixels being coloured and shaded as appropriate. The result of the rasterization process is a series of fragments for each of these primitives. A *fragment* is nothing more than a pixel with additional information about its colour, position and depth. The fragment's depth information is used to determine whether a particular pixel lies behind any of the other rasterized pixels. The matching pixel in the frame buffer is updated with the information carried by this fragment. This process of updating the pixels in the frame buffer with the fragments generated by the rasterizer is called *fragment processing*. The colour of fragments are manipulated using techniques such as texture mapping, bump mapping, texture filtering, environmental mapping, blending, per-fragment lighting, etc.

A.4 Programmable Pipelines

Today's graphics cards all have pipelines built into their graphics processing units. The operations that could be performed by earlier graphics cards were standardised by the device manufacturer with only a number of parameters and properties available for modification. Modern graphics cards allow for not only the modification of a large number of parameters, but also for complete control over the vertex and fragment

processors. These programmable vertex and fragment processors enable the real time rendering of various advanced techniques only previously achievable using large rendering farms or not even possible in real-time at all (Möller and Haines, 2002). Bump mapping (used for adding depth to pixels and thus creating a lighting-dependent bumpiness to a texture mapped surface) and environmental mapping (used for the generation of reflections by changing the texture coordinates based on the position of the camera) are just two examples of techniques only possible off-line in the past (Blinn, 1976), but that have become commonplace in the games of today (Percy et al, 1997). Figure A.4 shows a bump mapped surface with Figure A.5 showing the application of environmental mapping to simulate reflections on water.



Figure A.4 Bump mapping.

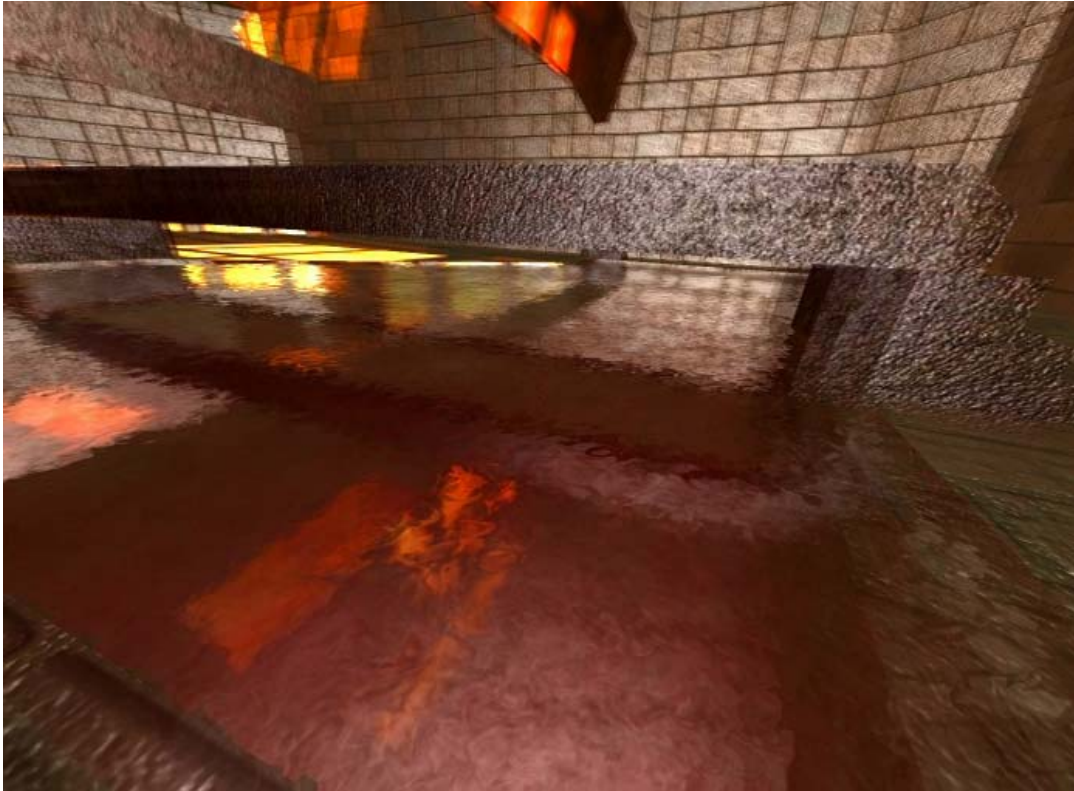


Figure A.5 Reflections on water using environmental mapping.

We will now look at Direct3D 10's programmable pipeline to fully understand the implication and use of programmable pipelines for the generation of advanced real-time graphical effects.

A.4.1 The Direct3D 10 Processing Pipeline

Each stage of the Direct3D 10 processing pipeline is configurable using the standard Direct3D application programming interface. The vertex shader, geometry shader and pixel shader are programmable using either Microsoft's proprietary High Level Shader Language (HLSL) or NVIDIA's C for Graphics (Cg). Each of these programmable processing units, including the pipeline processing states is discussed below. Figure A.6 illustrates the Direct3D 10 pipeline architecture.

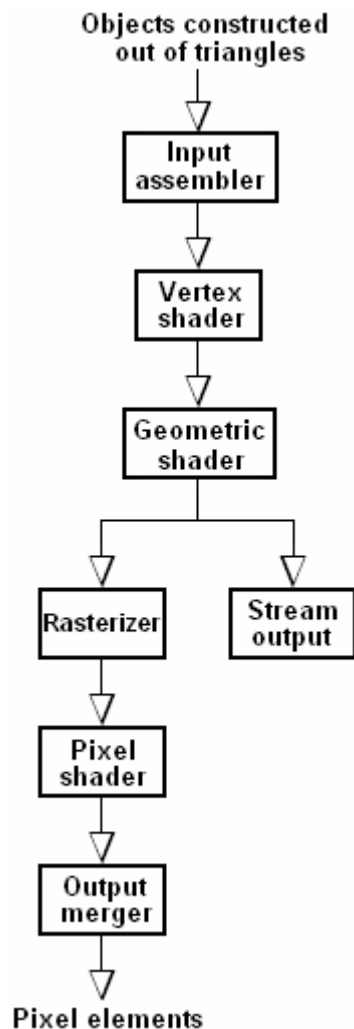


Figure A.6 Direct3D 10's programmable pipeline.

The Input-Assembler Stage

The first stage of the programmable pipeline, namely the *input assembler stage*, is responsible for propagating geometric input data consisting of points, lines and polygons to the rest of the pipeline. This pipeline stage assembles the input data into primitives, following this it forwards these assembled primitives to the next stage in the pipeline. For example, when data is received from some buffer it contains information about a vertex in three-dimensional space, the winding direction used for determining the vertex assembly order (either clockwise or counter-clockwise) and an identifier specifying the first vertex in a sequence of vertices. This information allows the input assembler to create primitive types supported by Direct3D. Figure A.7 illustrates how this information is used to create a supported primitive type.

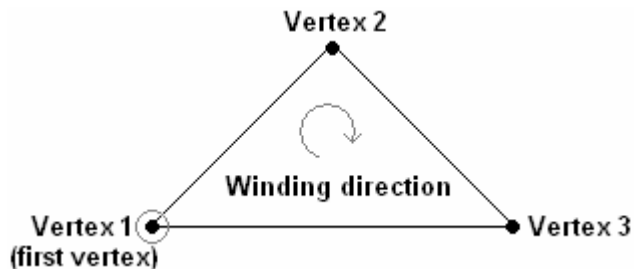


Figure A.7 Creating a triangle using three vertices, a clockwise winding direction and a vertex identifier indicating the first vertex in a set of three vertices.

The input assembler is also responsible for attaching *Shader System Values* for use by the shader core. These values (primitive id, vertex id, etc) lead to faster execution times by allowing the shader stages to ignore primitives that have already been dealt with.

Initialising the input assembler stage requires the specification of a vertex and optional index buffer that will be used for feeding the pipeline vertex data. The vertex buffer feeds the vertex data into the pipeline with the index buffer specifying indices for the vertex data stored in the vertex buffer. Creating a vertex buffer is relatively simple in Direct3D 10. We start by specifying the type of data that can be stored in the buffer (using the `D3D10_BUFFER_DESC` structure) followed by reading data into the buffer to initialise it (this data is specified using the `D3D10_SUBRESOURCE_DATA` structure). Once this is done we simply create the buffer using these descriptors. The `D3D10_BUFFER_DESC` structure describes the size of the buffer in bytes, the method how the buffer is to be read from and written to, the nature of the buffer (as a vertex buffer, index buffer, shader resource, etc), the kind of CPU access allowed (write, read, or 0 if no CPU access is necessary) and a flag to identify less regularly used options (such as resource sharing between various devices – 0 when not applicable). The `D3D10.h` header file specifies the `D3D10_BUFFER_DESC` structure as follows:

```
typedef struct D3D10_BUFFER_DESC {
    UINT ByteWidth;
    D3D10_USAGE Usage;
    UINT BindFlags;
    UINT CPUAccessFlags;
    UINT MiscFlags;
} D3D10_BUFFER_DESC;
```

The default values, including the alternatives, for the members of the `D3D10_BUFFER_DESC` structure are given in the following table:

Members	Flags
ByteWidth	Any number, for example: 64
Usage	<code>D3D10_USAGE_DEFAULT</code>

	(won't be read or written to by the CPU that often)
	D3D10_USAGE_IMMUTABLE (can't be written to by the CPU at all)
	D3D10_USAGE_DYNAMIC (buffer will be written to by the CPU at least once per frame)
	D3D10_USAGE_STAGING (read from and write to the GPU)
BindFlags	D3D10_BIND_VERTEX_BUFFER (specify the resource as a vertex buffer)
	D3D10_BIND_INDEX_BUFFER (specify the resource as an index buffer)
	D3D10_BIND_CONSTANT_BUFFER (specify the resource as a constant buffer which can only be updated completely, not partially, and which has a limit on the buffer's byte size)
	D3D10_BIND_SHADER_RESOURCE (specify the buffer as a shader resource)
	D3D10_BIND_STREAM_OUTPUT (specify the resource as an output buffer for the stream output stage discussed below)
	D3D10_BIND_RENDER_TARGET (specify the resource as a render target)
	D3D10_BIND_DEPTH_STENCIL (specify the resource as a depth-stencil buffer)
CPUAccessFlags	D3D10_CPU_ACCESS_READ (the buffer's contents can be read by the CPU)
	D3D10_CPU_ACCESS_WRITE (the CPU can change the buffer's contents directly instead of using the <code>UpdateSubresource ID3D10Device</code> interface)
MiscFlags	D3D10_RESOURCE_MISC_GENERATE_MIPS (specifies the creation of mipmaps for some texture resource using the <code>GenerateMips ID3D10Device</code> interface)
	D3D10_RESOURCE_MISC_SHARED (enables resource sharing between various devices)
	D3D10_RESOURCE_MISC_TEXTURECUBE (specifies the creation of a cube texture – a three dimensional texture in the shape of a cube constructed from six textures stored in a 2-D texture array)

Table A.1 Describing a buffer resource using the `D3D10_BUFFER_DESC` structure.

Before initialising the `D3D10_BUFFER_DESC` structure, we first have to specify the vertices for some geometric object. In this case our vertices will have both a spatial

location and a colour value (using the `D3DXVECTOR3` structure which has three members, an x-, y- and z-coordinate of a vector in three-dimensional space):

```
struct TriangleVertex
{
    D3DXVECTOR3 Location;
    D3DXVECTOR3 Colour;
};
```

We can now initialise the `D3D10_BUFFER_DESC` structure as follows for the specification of a vertex buffer description:

```
D3D10_BUFFER_DESC bufferDescription;

bufferDescription.Usage = D3D10_USAGE_DEFAULT;
bufferDescription.ByteWidth = sizeof(TriangleVertex) * 3;
bufferDescription.BindFlags = D3D10_BIND_VERTEX_BUFFER;
bufferDescription.CPUAccessFlags = 0;
bufferDescription.MiscFlags = 0;
```

Following this we create the vertex buffer using previously specified vertex data. The first step of this process is to specify an array of vertex data elements:

```
TriangleVertex array_of_vertex_data [] =
{
    D3DXVECTOR3( 0.0f, 1.0f, 1.0f ),
    D3DXVECTOR3( 0.0f, 0.0f, 0.5f ),
    D3DXVECTOR3( 1.0f, -1.0f, 1.0f ),
    D3DXVECTOR3( 1.0f, 0.0f, 0.0f ),
    D3DXVECTOR3( -1.0f, -1.0f, 1.0f ),
    D3DXVECTOR3( 0.0f, 1.0f, 0.0f ),
};
```

Next we have to initialise the `D3D10_SUBRESOURCE_DATA` structure. This data structure initialises a sub-resource using predefined data. A *sub-resource* is a portion of a resource that links back to the original resource data but with additional information about the resource so that the pipeline can easily access the data contained within this resource. The `D3D10_SUBRESOURCE_DATA` structure has three members, namely, a pointer to the data used for initialising the sub-resource, a value used for specifying the memory pitch in bytes required for two- and three-dimensional texture resources and the memory slice pitch associated with three-dimensional texture resources. The `D3D10.h` header file specifies this structure as follows:

```
typedef struct D3D10_SUBRESOURCE_DATA {  
    const void *pSysMem;  
    UINT SysMemPitch;  
    UINT SysMemSlicePitch;  
} D3D10_SUBRESOURCE_DATA;
```

We initialise the `D3D10_SUBRESOURCE_DATA` structure using the previously defined array of vertex data elements:

```
D3D10_SUBRESOURCE_DATA subresourceData;  
  
subresourceData.pSysMem = array_of_vertex_data;  
subresourceData.SysMemPitch = 0;  
subresourceData.SysMemSlicePitch = 0;
```

The final step is to create the vertex buffer. We use the `CreateBuffer ID3D10Device` interface to do this. This interface takes three parameters, the first being a pointer to the previously defined `D3D10_BUFFER_DESC` structure, the second a pointer to the `D3D10_SUBRESOURCE_DATA` structure with the third being the address of a pointer to the `ID3D10Buffer` interface used for controlling our buffer resource (be it either a vertex or index buffer). The `CreateBuffer ID3D10Device` interface is declared as follows in the `D3D10.h` header:

```
HRESULT CreateBuffer(  
    const D3D10_BUFFER_DESC *pDesc,  
    const D3D10_SUBRESOURCE_DATA *pInitialData,  
    ID3D10Buffer **ppBuffer  
);
```

We can now call the `CreateBuffer ID3D10Device` interface to create the vertex buffer:

```
ID3D10Device* g_id3dDevice;  
ID3D10Buffer* vertexBuffer[2] = {NULL, NULL};  
  
g_id3dDevice->CreateBuffer(&bufferDescription, &subresourceData, &vertexBuffer[0]);
```

Defining an index buffer is comparable to the creation of a vertex buffer, with the only difference being the specification of the `D3D10_BUFFER_DESC` structure's `BindFlags` member, for example:

```
D3D10_BUFFER_DESC indexBufferDescription;
```

```
indexBufferDescription.Usage = D3D10_USAGE_DEFAULT;
indexBufferDescription.ByteWidth = sizeof(TriangleVertex) * 3;
indexBufferDescription.BindFlags = D3D10_BIND_INDEX_BUFFER;
indexBufferDescription.CPUAccessFlags = 0;
indexBufferDescription.MiscFlags = 0;
```

We also have to specify an array containing index data. This array will be used to initialise the `D3D10_SUBRESOURCE_DATA` structure:

```
UINT array_of_index_data [] = {0, 1, 2, 3, 4};

D3D10_SUBRESOURCE_DATA indexSubresourceData;

indexSubresourceData.pSysMem = array_of_index_data;
indexSubresourceData.SysMemPitch = 0;
indexSubresourceData.SysMemSlicePitch = 0;
```

The index buffer is created using the `CreateBuffer` `ID3D10Device` interface:

```
ID3D10Buffer* indexBuffer = NULL;

g_id3dDevice->CreateBuffer(&indexBufferDescription,
                          &indexSubresourceData,
                          &indexBuffer);
```

With the input buffers specified and properly initialised, we create the *input-layout object* which will be used to control how vertex data is fed into the input-assembler stage (by directly describing the input-buffer data). The type of the input vertex data is identified and checked against shader parameter types ensuring both type compatibility and that the needed shader data is actually stored in the buffer. We create the input-layout object using the `CreateInputLayout` `ID3D10Device` interface via the specification of five parameters. The first parameter is an array of the input-assembler stage input data type described using the `D3D10_INPUT_ELEMENT_DESC` structure. The `D3D10_INPUT_ELEMENT_DESC` structure is defined as follows in the `D3D10.h` header file:

```
typedef struct D3D10_INPUT_ELEMENT_DESC {
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset;
    D3D10_INPUT_CLASSIFICATION InputSlotClass;
```

```

    UINT InstanceDataStepRate;
} D3D10_INPUT_ELEMENT_DESC;

```

This structure gives a description of each input assembler stage element, specifically; the High Level Shader Language (HLSL) semantic name of the element, the element's semantic index used when more than one element with the same semantic name exists, the element's data type, an integer value used for specifying the input-assembler's input slot (described below), the byte offset used to set the location of the element in the input slot (counting in bytes from the beginning of the input slot), the input data class (either vertex data using the `D3D10_INPUT_CLASSIFICATION` enumeration with the constant set to either `D3D10_INPUT_PER_VERTEX_DATA` for per-vertex input data, or `D3D10_INPUT_PER_INSTANCE_DATA` for per-instance input data) and the data step rate controlling the number of instances of one element to draw (using the per-instance input data) before moving on to the next buffer element – must be set 0 for elements containing per-vertex data. Using the `D3D10_INPUT_ELEMENT_DESC` structure, we can specify a vertex buffer containing two vertex-data elements as follows:

```

D3D10_INPUT_ELEMENT_DESC input_layout_description[] =
{
    {L"POSITION", 0, DXGI_FORMAT_R32G32B32_UINT, 0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0},
    {L"COLOR", 0, DXGI_FORMAT_R32G32B32_UINT, 1, 6, D3D10_INPUT_PER_VERTEX_DATA, 0},
};

```

Data is fed into the input-assembler stage through a number of units referred to as *input slots*. Each of these input-assembler input slots, shown in Figure A.8, are used as storage for a vertex buffer, thus storing input data.

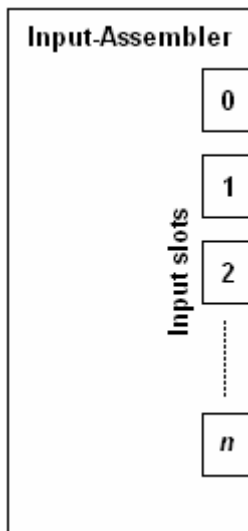


Figure A.8 The input-assembler's input slots.

The second parameter of the `CreateInputLayout ID3D10Device` interface is an integer value specifying the number of input-data types making up the input-elements array. The third parameter is a pointer to the compiled shader code with the fourth parameter specifying the byte size of this compiled shader code. The final parameter is a pointer to the input-layout object that will be used as output. This `CreateInputLayout ID3D10Device` interface is defined as follows in the `D3D10.h` header file:

```
HRESULT CreateInputLayout (  
    const D3D10_INPUT_ELEMENT_DESC *pInputElementDescs,  
    UINT NumElements,  
    const void *pShaderBytecodeWithInputSignature,  
    SIZE_T BytecodeLength,  
    ID3D10InputLayout **ppInputLayout);
```

We can now bind this newly created input-layout object to the input-assembler stage, after which we can call the draw functions. This object binding is done using the `IASETVertexBuffers` and `IASETInputLayout ID3D10Device` interfaces. The `IASETVertexBuffers` interface binds a vertex buffer array to the input-assembler stage by specifying the input slot, the total number of buffers in the vertex buffer array, a pointer to the vertex buffer array, a pointer to an array containing values indicating the byte size of elements to be read from the vertex buffer (referred to as stride values) and a pointer to an array containing so called offset values (with one offset value representing the number of bytes to be read from the first element stored in the vertex buffer to the element being accessed). This `IASETVertexBuffers ID3D10Device` interface is defined as follows in the `D3D10.h` header file:

```
void IASETVertexBuffers(UINT StartSlot, UINT NumBuffers,  
    ID3D10Buffer *const *ppVertexBuffers,  
    const UINT *pStrides,  
    const UINT *pOffsets);
```

The `IASETInputLayout` interface, taking a pointer to the input-layout object, is responsible for binding this object to the input-assembler stage. The following code sample illustrates this process:

```
UINT start_input_slot = 0;  
UINT number_buffers_in_array = 1;  
UINT offset_value = 0;  
UINT stride_value = sizeof(TriangleVertex);  
  
g_id3dDevice->IASETVertexBuffers(start_input_slot,  
    number_buffers_in_array,
```

```
&vertexBuffer,  
&stride_value,  
&offset_value);
```

The input-layout takes a pointer to the `ID3D10Device` object:

```
ID3D10InputLayout* inputLayoutObject = NULL;  
  
g_id3dDevice->IASetInputLayout(inputLayoutObject);
```

The only remaining step is to specify the assembling of vertices into primitives and to send these primitives (controlling the rendering of vertex data to the screen) to the next step of the pipeline. This is done using the `IASetPrimitiveTopology` `ID3D10Device` interface. This interface takes one parameter, namely the primitive type specified using the `D3D10_PRIMITIVE_TOPOLOGY` enumerator. For example, the following code specifies the primitive type as a list of lines:

```
g_id3dDevice->IASetPrimitiveTopology( D3D10_PRIMITIVE_TOPOLOGY_LINELIST);
```

Table A.2 lists possible primitive types:

Constant	Description
<code>D3D10_PRIMITIVE_TOPOLOGY_UNDEFINED</code>	A primitive topology is not specified for the Input-assembler stage.
<code>D3D10_PRIMITIVE_TOPOLOGY_LINELIST</code>	The vertex data is interpreted as a list of lines.
<code>D3D10_PRIMITIVE_TOPOLOGY_LINELIST_ADJ</code>	The vertex data is interpreted as a list of lines with adjacency data.
<code>D3D10_PRIMITIVE_TOPOLOGY_LINESTRIP</code>	The vertex data is interpreted as a line strip.
<code>D3D10_PRIMITIVE_TOPOLOGY_LINESTRIP_ADJ</code>	The vertex data is interpreted as a line strip with adjacency data.
<code>D3D10_PRIMITIVE_TOPOLOGY_POINTLIST</code>	The vertex data is interpreted as a list of points.
<code>D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST</code>	The vertex data is interpreted as a list of triangles.
<code>D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ</code>	The vertex data is interpreted as a list of triangles with adjacency data.
<code>D3D10_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP</code>	The vertex data is interpreted

	as a triangle strip.
D3D10_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP_ADJ	The vertex data is interpreted as a triangle strip with adjacency data.

Table A.2 Specifying a primitive type using the `D3D10_PRIMITIVE_TOPOLOGY` enumerator.

We can now draw these pipeline bound primitives using various `ID3D10Device` functions such as `Draw`, `DrawAuto`, `DrawIndexed`, `DrawInstanced` and `DrawIndexedInstanced`.

The Vertex-Shader Stage

Per-vertex operations are performed during this pipeline processing stage. Examples of such operations include per-vertex lighting, texture sampling operations, geometric transformations, etc. Per-vertex lighting allows us to specify distinct light sources, including the interaction of these light sources with adjacent surfaces. These interactions and reflections are considered on a per-vertex basis with the lighting values between vertices being approximated. This stage takes one vertex as input, modifies it according to some predefined operation and outputs it for further processing. There might also be cases where no vertex processing is required, leading to the definition of a pass-through vertex shader. This *pass-through vertex shader* forwards the input vertex data to the geometry-shader stage unmodified.

Input vertex data generally consist of anything from one to sixteen 32-bit vectors made up of one to four elements each. The input assembler basically feeds two data elements into the vertex-shader stage, namely; the vertex ID and the instance ID. These IDs are generated by the graphics hardware and can only be handled during this pipeline stage.

The Geometry-Shader Stage

Primitives such as vertices, lines and polygons are processed during this pipeline stage. The geometry-shader stage takes these primitives as input, and processes them based on some programmatically defined algorithm, forwarding these newly modified or, in some cases, newly generated primitives to either the stream-output stage or rasterizer stage. The geometry-shader stage takes full primitives as input, for example; lines consisting of two vertices, quads constructed out of four vertices, etc (Stam and Loop, 2003). This is in contrast with vertex shaders which only accept a single vertex as input.

One useful feature of the geometry-shader is its ability to handle edge-adjacent primitives. For example, say we have a quad as input; then the vertex data of all

primitives adjacent to the quad can also be read as input. Figure A.9 shows such a quad with four adjacent quads.

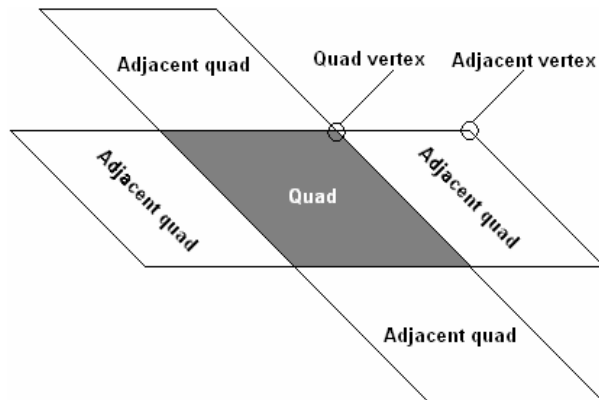


Figure A.9 A quad with edge-adjacent primitives.

The geometry-shader's generated primitives are returned as an output stream object. This output stream can be declared as a `LineStream` (creating a line strip output topology), `TriangleStream` (creating a triangle strip output topology) or `PointStream` (creating a point list output topology) based on the original primitive object type. We create a primitive strip by appending output vertices using the `Append` interface method. The appending of vertices is necessary since the geometry-shader only outputs one vertex data element at a time – requiring this vertex data to be reconstructed into primitives. The `RestartStrip` method is used to terminate the current primitive strip construction process, signalling the geometry-shader to start the creation of a new primitive strip. The following non-functional code sample shows the creation of a `TriangleStream` output object via the declaration of a geometry-shader.

We start by setting the maximum number of vertices to output using the `MaxVertexCount` attribute type (causing the geometry-shader to terminate once the specified number of vertices has been generated):

```
[MaxVertexCount(6)]
```

Next we declare the geometry-shader, `GS_sample`, to take a triangle strip or triangle list (`triangle float4 inputPar[3]`) as input; with a `TriangleStream` object as output (the `inout` keyword declares the stream object, `outputPar`, as both an input and output):

```
void GS_sample(triangle float4 inputPar [3], inout TriangleStream<float2> outputPar)
{
    //function body
    //e.g. using Append and RestartStrip:
    outputPar.Append(...);
}
```

```

outputPar.RestartStrip();
}

```

Modern day computer games are increasingly making use of geometric shaders, mostly due to the exponential advances being made in graphics hardware and the power given to developers in controlling this hardware at a functional level using shading languages such as HLSL, Cg and the OpenGL Shading Language (GLSL). Examples of effects derived from programming DirectX 10's geometry-shader include shadow volume generation, fur animation, advanced dynamic particle systems, cube mapping, point sprite expansion and various other per-primitive operations.

Stream Output Stage

The stream output stage streams primitives from the geometry-shader stage to predefined buffers in system memory or memory present on the graphics card. This data can either be fed back into the input-assembler stage or alternatively loaded directly into shaders via load functions, or circulated to the CPU, for example (Figure A.10). The adjacency data associated with primitives outputted by the geometry-shader stage is discarded when output is directed to the stream output stage. Triangle and line strips are also converted to triangle and line lists when streamed to the buffer resources in memory.

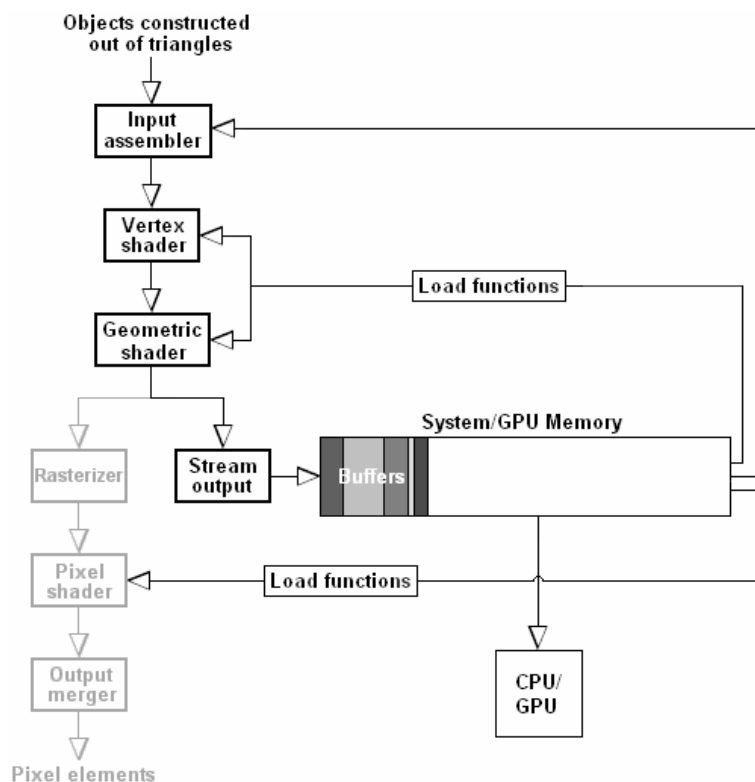


Figure A.10 Streaming of data to predefined buffers in system/GPU memory.

One or multiple buffers can be linked to the stream output stage. When one buffer is linked, then anything from 1 to 64 scalar per-vertex data elements can be written to the buffer (assuming a total size less than 257 bytes for the per-vertex output data elements). The use of multiple buffers, with each catching a single per-vertex data element, enables us to output data to a maximum of four buffers concurrently. When using multiple buffers it is not required for all the buffers to have the same size. The output of data to these varying sized buffers terminate the moment the smallest buffer is full (unable to receive any more primitives as input).

The Pixel-Shader Stage

The rasterizer stage rasterizes primitives produced by the geometry-shader stage into pixels via the interpolation of vertex values for representation in the frame buffer and for subsequent output to a monitor. The shading and colour of these pixel values need to be calculated so that each primitive are correctly rendered to the display device. The rasterizer stage calls the pixel-shader stage for the computation of these per-pixel values. Various per-pixel shading techniques such as lighting, fog, bump mapping, shadows, distortion effects and shading are performed during this stage (Legakis, 1998). In addition to these effect-based per-pixel techniques, the pixel shader is also used for implementing level-of-detail algorithms and during the process of anisotropic filtering crucial for enhancing the image quality of distant located textures.

Programs defining pixel-shader operations are called shader programs and can be written in any of the following languages: Assembly, Cg, HLSL or GLSL. These programs normally take colour values, the interpolated per-vertex data produced by the rasterizer stage, and some user defined variables as input, producing the final pixel values that are forwarded to the output-merger stage.

The Output-Merger Stage

This final stage of the Direct3D 10 programmable pipeline combines both the output generated by the pixel-shader stage with depth- and stencil buffer values to produce the final pixel colour and shading values. The output-merger stage is directly responsible for determining the visibility of pixels based on the process of depth testing. The blending of pixel data (combining two or more pixel colour values), in addition to depth- and stencil testing, is also controlled during this pipeline stage.

Shaders

A shader is a grouping of instructions processed by the graphics accelerator to perform some form of special effect or rendering. The previous section presented the concept of programmable pipelines, in particular focusing on the Direct3D 10 and OpenGL processing pipelines. An application program allowing direct interaction with these previously discussed programming pipelines is called a *shader*. These shader programs, written in a shading language such as NVIDIA's Cg or Microsoft's High Level Shader Language, control the movement, composition, form and appearance of objects through direct manipulation of the graphics processing unit is programmable pipelines.

The instructions listed in a shader program are executed at a specific point in the rendering pipeline – thus leading to user-defined manipulation of vertex or pixel data, for example. More specifically, three types of shader programs can be written, namely, vertex shaders, pixel shaders and geometry shaders.

Vertex shaders, operating on vertex data, are executed as part of the graphics pipeline's geometric stage and are used to alter the geometric parameters (shape) of an object. A vertex shader program is fundamental for certain special effects such as grass blowing in the wind where the real time manipulation, transformation and displacement of per-vertex material attributes are necessary. The vertices produced by this shader are forwarded as input to a geometry shader.

Geometry shaders are executed just prior to the rasterizer and stream output pipeline stages. These shaders group numerous vertices into a geometric object that can be modified by a pixel shader program. Geometry shaders are extremely important in the detection of silhouetted-edges and shadow volume extrusion. These shaders, performing per-primitive computations, are also vital in the generation of new primitives. The primitives generated by the geometry shader stage are rasterized into fragments during the pipeline's rasterizer stage. These fragments are then sent to the pixel shader as input.

Pixel shaders, also known as *fragment shaders* and performing per-pixel processing, operate on the discrete pixels of a primitive, applying some effect to a primitive (such as bump mapping, shadowing, fog, etc) during the pixel shader stage. Per-pixel lighting and shadowing has greatly contributed to the realism of modern computer games. Examples of effects made possible through this form of per-pixel processing include texture blending, environmental mapping, normal mapping, real-time shadows (stencil shadow volumes) and reflections (Levoy and Hanrahan, 1996).

These three types of shaders are unified by the Direct3D 10 architecture – known as Shader Model 4.0. *Unified shaders* provide the application programmer with a uniform instruction set independent of whether a pixel shader or vertex shader is being implemented. This unified architecture is made possible through Windows Vista's Windows Display Driver Model and the coupled DirectX 10 API. Previous architectures required different instruction sets for both pixel and vertex shaders due to specific hardware architectural requirements. By unifying the independent shader instruction sets, GPU programming has become much more flexible. This unified model also allows workload sharing amongst the various pipeline processors, for example, when the GPU is mainly performing basic geometry rendering with little or no per-pixel processing being done, then the pixel shader can be assigned vertex processing. The first GPU offering support for this unified shader model was NVIDIA's GeForce 8 series – specifically the GeForce 8800 GTX and GTS.

The term used to describe this unified shader architecture, Shader Model 4.0, encapsulates the features offered by the specific shader version in question. For example, Shader Model 3.0 (as supported by Direct3D 9.0c) limits the number of executing instructions to 65536 while Direct3D 10's Shader Model 4.0 allows for an unlimited number of executing instructions. Shader Model 2.0 (the original Direct3D 9.0 shader specification) limits the number of executing instructions to 32 texture instructions and 64 arithmetic instructions. The version number of instructions is specified in terms of the shader's version number (`ps_mainVersion_subVersion` for pixel shaders and `vs_mainVersion_subVersion` for vertex shaders). For example, a vertex shader based on Shader Model 3.0 (DirectX 9.0c) will be declared as `vs_3_0`, a DirectX 9.0b Shader Model 2.0 pixel shader as `ps_2_b`, with a Shader Model 4.0 pixel shader declared as `ps_4_0`. NVIDIA's GeForce FX series of GPUs provide an optimised model for Shader Model 2.0 and we can thus define a vertex shader based on this model as `vs_2_a`.

The capabilities of shader programs are heavily dependent on the available graphics hardware. Older graphics hardware such as first-generation GPUs (NVIDIA's RIVA TNT2 and ATI's Rage series implementing the DirectX 6 feature set) were only capable of accelerating texture mapping operations as well as the rasterization of certain primitives such as triangles. These GPUs alleviated the CPU from updating individual pixels but vertex transformations such as rotation, translation and scaling were still CPU dependant. These GPUs, although slightly configurable, were not programmable.

The second-generation of GPUs, introduced in 1999/2000 with the release of NVIDIA's GeForce 256 GPU and also including the GeForce2 and ATI's Radeon 7500, relieved the CPU from 3-D vertex transformations and lighting computations. Both the OpenGL and DirectX 7 APIs supported these hardware vertex transformations, however, although highly configurable in the sense of offering support for certain effects such as

cube mapping for textures and per-pixel colouring, these GPUs were still not strictly speaking programmable.

The first truly programmable GPUs were NVIDIA's third-generation GeForce3, GeForce4 Ti and ATI's Radeon 8500 series. These GPUs offered programmable vertex pipelines, thus allowing an application program to control vertex transformations and lighting. These GPUs also featured a higher level of per-pixel configurability, although not yet offering pixel pipeline programmability. DirectX 8 and the `ARB_vertex_program` OpenGL extension allowed access to the vertex programmability offered by these GPUs. Pixel shaders could be written using the DirectX 8 pixel shader functionality and numerous OpenGL extensions. These pixel shaders were obviously nothing as powerful as today's pixel shader programs, and were based on configuring the pixel pipeline, rather than freeing the CPU of pixel-shading operations.

Both per-vertex and per-pixel programmability have been available since the release of NVIDIA's GeForce FX and ATI's Radeon 9700 family of GPUs. Application developers were, with the release of these GPUs, for the first time able to assign the GPU for both vertex transformations and pixel operations. With these operations offloaded to the GPU, the CPU is free to perform other calculations. The DirectX 9 API and several OpenGL extensions give access the pixel and vertex programmability offered by these GPUs. A vertex shader replaces the configurable fixed-function operations performed by the vertex processor with instructions defined by the shader along with a pixel shader executing after the rasterizer stage. This pixel shader takes the fragments processed by the fragment processor/pixel shader stage as input, performing some operation on them. Fragments are processed based on some configurable fixed function in the absence of a pixel shader program.

Table B.1 highlights some key features introduced with certain milestone GPU releases as well as their respective DirectX and OpenGL version support.

GPU	Main Feature(s)	API support
- NVIDIA RIVA 128	- Basic vertex acceleration.	DirectX 5, OpenGL 1.0.
- NVIDIA RIVA TNT - NVIDIA RIVA TNT2 - ATI Rage 128	- Multitexturing (applying more than one texture to a polygon, e.g. graffiti art or 'bullet holes' on a textured wall).	DirectX 6, OpenGL 1.1.
- NVIDIA GeForce 256 - NVIDIA GeForce2 - ATI R100 (Radeon 32, 64, 7000 and 7500)	- Hardware Transformations, Clipping and Lighting. - Cube mapping. - Fixed-function vertex processing. - Register combiners.	DirectX 7, OpenGL 1.2 (ATI supporting OpenGL 1.3)
- NVIDIA GeForce3	- Quadtexturing (using four pixel	DirectX 8,

	<p>pipelines for the rendering of four independently textured pixels or alternatively two multitextured pixels)</p> <ul style="list-style-type: none"> - Texture shaders. - Shader Model 1.1. - ARB_vertex_program (OpenGL extension for vertex shaders on both ATI and NVIDIA chipsets). 	OpenGL 1.4.
<ul style="list-style-type: none"> - NVIDIA GeForce4 Ti - ATI Radeon R200 (Radeon 8500 to 9250) 	<ul style="list-style-type: none"> - Hardware anti-aliasing. - Pixel Shader 1.2, 1.3 or 1.4. - Vertex Shader 1.1. - ATI_fragment_shader (OpenGL extension for fragment shaders on ATI cards only). 	DirectX 8.1, OpenGL 1.4.
<ul style="list-style-type: none"> - NVIDIA GeForce FX - ATI Radeon R300 (Radeon 9500 to 9800 XT and including Radeon X1050) 	<ul style="list-style-type: none"> - Full support for vertex and fragment shader programs. - Floating-point pixel processing. - Shader Model 2.0, 2.0a or 2.0b. - OpenGL Shading Language. 	DirectX 9.0b, OpenGL 1.4 (NVIDIA chipsets featured limited support for OpenGL 2.0 with ATI chipsets offering full support).
<ul style="list-style-type: none"> - NVIDIA GeForce 6 - ATI Radeon R500 (Xbox 360 Xenos, Radeon X1300 to Radeon X1950 XTX) 	<ul style="list-style-type: none"> - Hardware accelerated transparency. - Scalable Link Interface (SLI – parallel graphics processing using two or more graphics accelerators interlinked). - Shader Model 3.0. - OpenGL Shading Language Improved. 	DirectX 9.0c, OpenGL 2.0.
<ul style="list-style-type: none"> - NVIDIA GeForce 7 	<ul style="list-style-type: none"> - High Dynamic Range Lighting. 	DirectX 9.0c, OpenGL 2.0.
<ul style="list-style-type: none"> - NVIDIA GeForce 8 - Radeon R600 (Radeon HD 2400 to Radeon HD 2900 XT) 	<ul style="list-style-type: none"> - Unified Shaders. - Shader Model 4.0. 	DirectX 10, OpenGL 2.1.
<ul style="list-style-type: none"> - NVIDIA GeForce 9/100/250/260-295 	<ul style="list-style-type: none"> - Atomic functions (thread-safe) - Coverage Sample AA - 128 bit OpenEXR 	DirectX 10, OpenGL 3.3.
<ul style="list-style-type: none"> - NVIDIA GeForce 210/220/240/300 	<ul style="list-style-type: none"> - Shader Model 4.1 	DirectX 10.1, OpenGL 3.3.
<ul style="list-style-type: none"> - NVIDIA GeForce 400/500 	<ul style="list-style-type: none"> - Shader Model 5.0 	DirectX 11, OpenGL 4.1, OpenCL 1.0

Table B.1 Features introduced by selected GPUs and DirectX and OpenGL versions.

B.1 The Hardware Graphics Pipeline Revisited

We previously described a pipeline as a series of parallel stages with each stage processing the output of the previous stage, in turn sending its output to a successive stage, and so forth. The graphics pipeline consists of a number of stages such as vertex processing, clipping, rasterization and fragment processing. These stages are responsible for converting some geometrically defined scene into a two-dimensional image (pixel elements) via a number rendering stages – each physically organised as a pipeline processing unit. We will now revisit our previous graphics pipeline architecture discussion, expanding on it by focussing more on the programmable graphics pipeline’s physical (hardware-level) organisation.

A modern-day GPU is sent a grouping of vertices organised into a geometric primitive such as a sequence of points, lines or a triangle, for example. Each of these vertices has a number of attributes. Attributes can range from the vertex’s individual colour value, its texture coordinates, a normal vector used during lighting calculations to spatial coordinates used for the positioning of the vertex. A generic graphics hardware pipeline is show in Figure B.1.

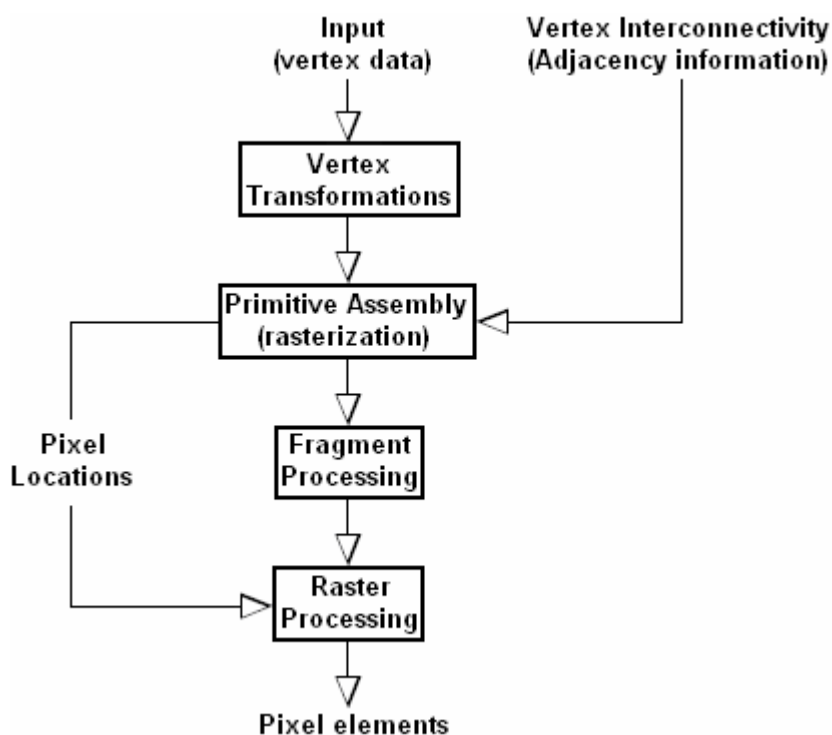


Figure B.1 A generic graphics hardware pipeline.

The vertex transformation stage performs a series of operations on each of the vertices sent to the GPU for processing. Operations include the transformation of a vertex’s

coordinate system into one that can be used by the rasterizer, per-vertex lighting, colouring and the generation of texture coordinates, etc.

The primitive assembly and rasterization stage assembles the vertices being passed from the vertex transformation stage into geometric primitives. The type of assembled primitive (line, polygon, triangle, etc) depends on the primitive topology data accompanying a set of vertices. Clipping to the visible view frustum is performed during this stage, resulting in the elimination of any unnecessary primitives that would not be visible to the viewer or camera (Liang and Barsky, 1984). The rasterization stage also eliminates polygons or surfaces pointing away from the camera or viewer (vertex-by-vertex culling). Following these operations, primitives are rasterized into pixels for representation in the frame buffer (Sutherland and Hodgman, 1974). Rasterization is performed according to a specific set of rules defined for each of the primitive topologies. The rasterization stage produces a set of pixels, each one mapped to a specific location, as well as a set of fragments (previously defined as a pixel with additional information about its colour, position and depth). Building on our previous definition we can now define a fragment as a state necessary for the update of a specific pixel in the frame buffer. During the rasterization process geometric primitives are broken down into pixel-sized fragments. Each fragment holds information about the pixel's location, depth, colour and texture coordinates. This information is then used to update a matching pixel in the frame buffer.

With a primitive successfully rasterized into a series of fragments, we can move on to the fragment processing stage. Fragment processing, as previously explained, is the process of updating pixels in the frame buffer with the fragments generated during the rasterization stage. The fragment processing unit is responsible for setting the colour values of fragments, their texturing as well as the interpolation of fragment parameters. These operations are modified and/or combined for numerous texturing effects such as bump mapping, texture filtering, blending, environmental mapping and so forth. Apart from calculating the fragment's final colour value, this pipeline stage can also discard a fragment based on some calculation or predefined parameter, hence resulting in the corresponding frame buffer pixel not being updated.

The final number of fragment centric operations, based on the functionality of Direct3D and OpenGL, are performed during the raster processing stage. These operations, such as depth testing (the removal of hidden surfaces), blending, stencil testing for the generation of stencil shadow volumes and stencil based reflections, etc, are performed prior to the frame buffer update. A number of tests are conducted during this pipeline stage; for example, a scissor test culls all the fragments located outside a user-specified rectangle positioned within the render target area, with an alpha test determining whether fragments are written to the render target area based on some predefined alpha-test function. A fragment is discarded whenever any of these tests fail. When passing a specific test, one of the pixel's property values (such as depth for depth

testing) is updated with that of the fragment. The blending operation stage reads the fragment's colour value and combines it with the colour value of the matching pixel. We can also dither the colour values of fragments and pixels to create the illusion of colour depth in low-colour images by approximating colours not available in the palette through the diffusion of the available palette's colour values. The final operation is to write the new blended/dithered fragment colour value out to the appropriate pixel in the frame buffer. This raster processing stage, consisting of a series of pipeline stages (raster operations and tests), is shown in Figure B.2.

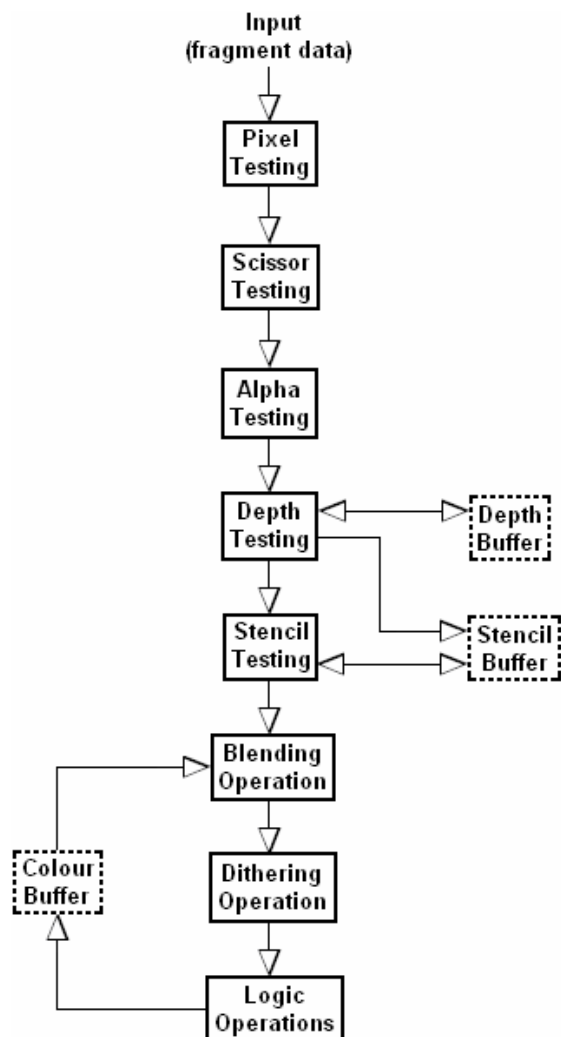


Figure B.2 Direct3D and OpenGL raster processing operations.

B.2 The Programmable Graphics Pipeline Revisited

This section extends the previous discussion of the Direct3D processing pipeline by investigating the underlying hardware configuration that makes the pipeline stages of a GPU programmable. Previous generation GPUs have separate vertex and pixel shader

processing units. The GeForce 8 GPU (and better) does not follow this approach, rather offering eight shader units, with none of them limited to vertex or pixel processing. This architectural change is the product of recognising that the future of GPU design lies with programmable processing. By unifying the shaders we're not just only able to use the same instruction set for both pixel and vertex shaders or to enable workload sharing amongst these pipeline processors, but this new architecture also makes it easier to extend our current shader model with future shader types. As illustrated in Table B.1, GPU architecture has evolved from supporting configurable vertex and fragment processors, to programmable vertex processors, then fully programmable vertex and fragments processors to the current unified architecture. Extending the generic graphics hardware pipeline, we can show both vertex and fragment processing units as simple add-ons to this generic pipeline (Figure B.3).

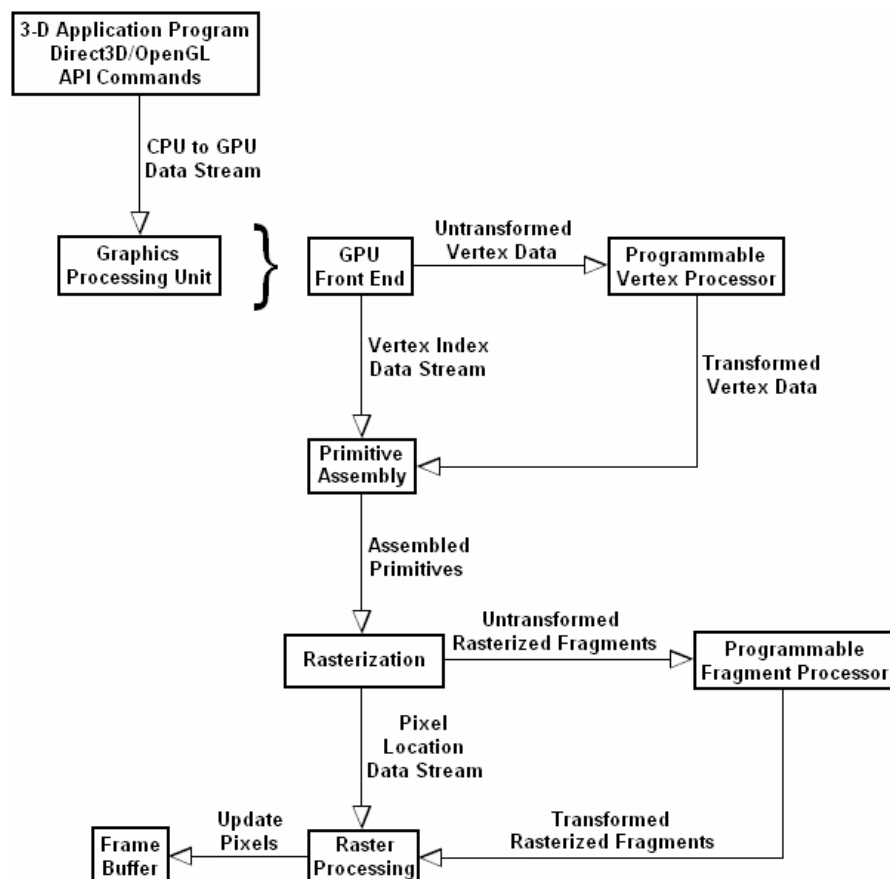


Figure B.3 Example of a hardware programmable graphics pipeline.

The unified shader architecture considered, vertex and fragment processing can still be broken down into logical programmable units; with a programmable vertex processor, the processing unit responsible for the execution of a HLSL, Cg or GLSL vertex program and a programmable fragment processor, the processing unit tasked with execution of a HLSL, Cg or GLSL fragment program.

Focussing on the programmable vertex processor, we can summarise its functionality into a number of stages. The first stage feeds vertex attributes such as coordinates, colour values and depth information into the vertex processor for processing. These vertex attributes are stored in the vertex attribute register banks. Vertex shaders actually make use of a several registers for the storage of position, data and colour data, for example. The vertex program, consisting of a sequence of instructions, is stored in memory. The vertex processor accesses this program, decoding one instruction at a time until the program terminates. Results generated from computations, the transformed vertex data, are stored in the output result registers with intermediate data, still being read by instructions, stored in the temporary register banks. Figure B.4 shows the classic flow of control for a programmable vertex processor.

Programmable fragment processors are extremely useful for manipulating texture coordinates as well as to set the final colour of a pixel. These processors also support several of the vector math operations performed by vertex processors. For example, a fragment processor can be programmed to read the texture coordinates of a textured image and to subsequently perform some operation on these values – returning a filtered sample of the texture. Similar to a vertex processor, fragment processors operate by executing a set of instructions stored in a program file – the fragment program. These instructions are executed until the fragment program terminates (when there aren't any more instructions to fetch). The fragment program reads untransformed interpolated fragments as input, storing these values in input register banks. Results generated from applying the specified instructions on input data are stored in the output registers. Intermediate data, just as with vertex processing, is stored in the temporary register banks. The output values can range from a fragment's new colour to a transformed depth value.

A texture is nothing more than a two-dimensional array consisting of colour values with each of these colour values referred to as a texel, or texture element. Each texel, being an element in this colour array, is thus assigned a unique address in the texture (simply a column and row value). Fragment processors generally include a *texture fetch instruction*. This instruction is used to compute the address of a texture, fetch texture elements, determine its Level-of-Detail and to perform texture filtering. Examples of texture filtering include nearest-point sampling, linear texture filtering, anisotropic texture filtering, bilinear filtering and filtering via mipmaps. Figure B.5 illustrates the flow of control for a typical programmable fragment processor.

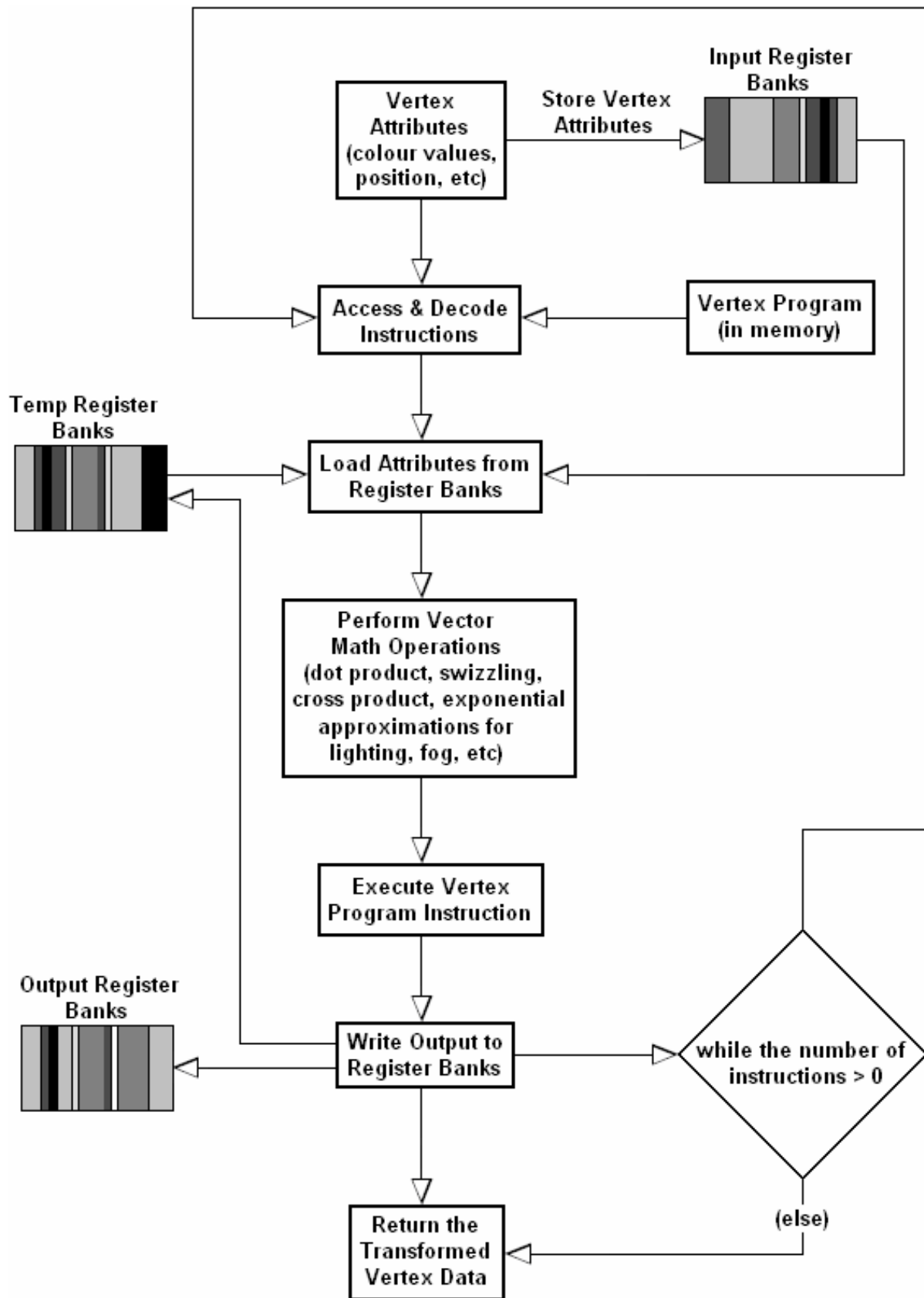


Figure B.4 Flow of control for a programmable vertex processor.

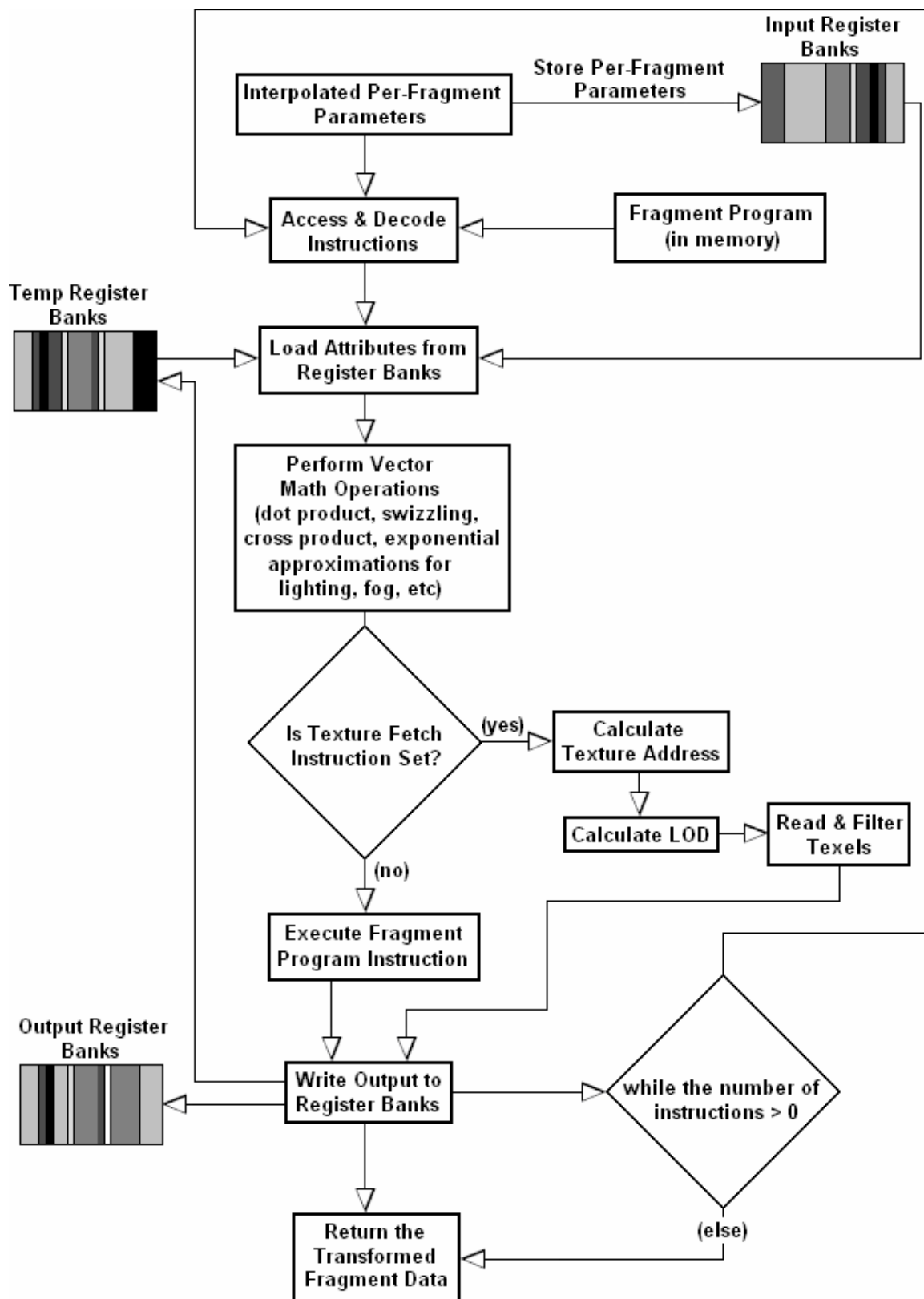


Figure B.5 Flow of control for a programmable fragment processor.

B.3 High Level Shader Language (HLSL)

Microsoft's High Level Shader Language is a proprietary Direct3D shading language analogous to NVIDIA's Cg. The Direct3D 10 High Level Shader Language allows for the creation of three types of shader programs, namely, vertex shaders, geometry shaders and pixel shaders. Similar to Cg, HLSL shaders can be compiled either statically or dynamically, depending on the preference of the developer and intended application for the shader.

As mentioned, Direct3D 10 shaders are unified to provide the application programmer with a uniform instruction set independent of whether a pixel, vertex or geometric shader is being implemented. These different shaders, offering the same core functionality, are implemented by the Shader Model 4.0 common shader core. Building on the core functionality, each shader implementation offers its own unique functionality such as stencilling done via pixel shaders or the generation of new primitives and the manipulation of 3-D models on a per-primitive basis by a geometric shader. This common shader core data-flow is shown in Figure B.6.

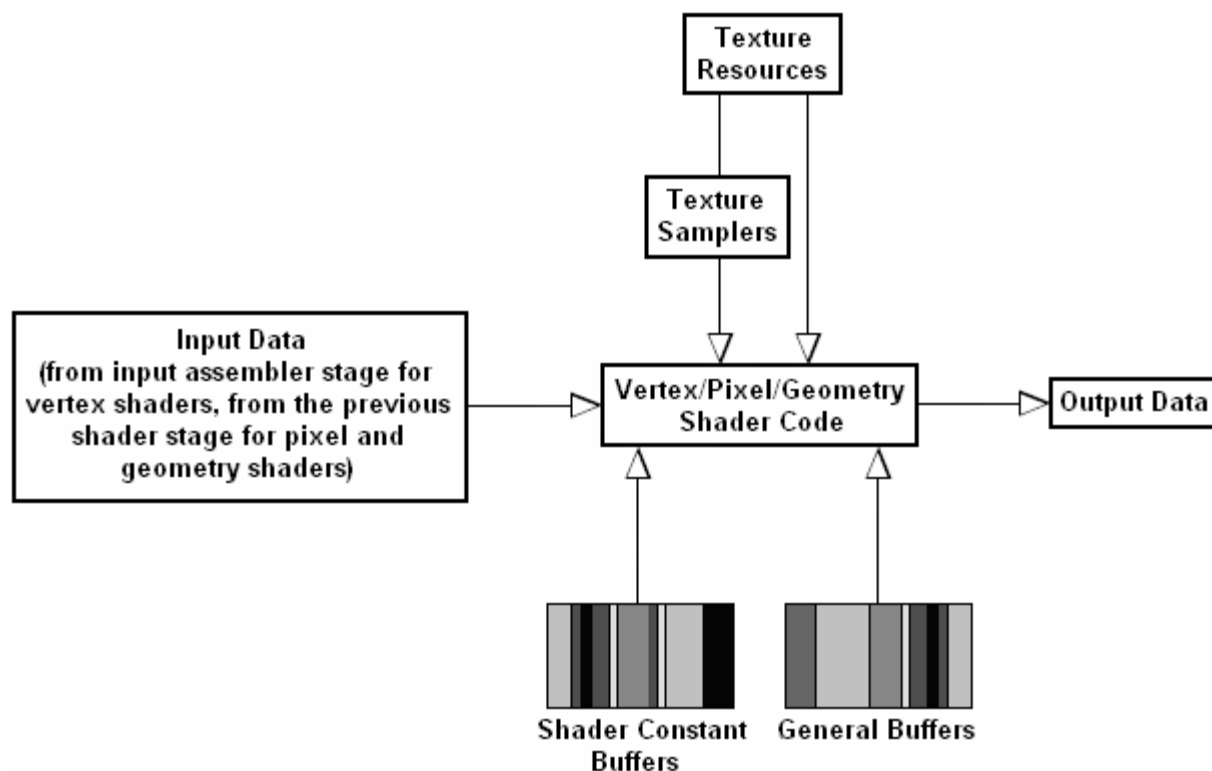


Figure B.6 Common shader core architecture.

The stages given in the above depicted data-flow model can be summarised as follows:

- 1) Input data is sent to the vertex, pixel or geometry shader for processing with the vertex shader receiving data from the input assembler stage and the pixel and geometry shaders receiving their input data from the previous shader stage.
- 2) The shaders can now perform some arithmetic or flow control operation on the read data.
 - a. Texel data is either directly read without any filtering or sampling using the `Load` HLSL function or alternatively filtered and sampled by binding up to 16 HLSL samplers to the shader.
 - b. General buffers are also accessed from system memory, allowing the shader program to bind up to 128 texture elements and buffer resources to the shader.
 - c. Shader constant buffers can also be bound to a shader stage. These buffers are frequently updated by the CPU and are larger in size and layout than the general buffers.
- 3) The output generated by the shader code is passed to the next stage in the graphics pipeline.

B.3.1 The HLSL Compiler

HLSL programs have to be compiled into a GPU executable form. Compilation is based on the translation of a vertex, pixel or geometry shader program into a form readable by Direct3D. This translation of the original HLSL program is sent to the Direct3D API driver which converts it to instructions that can be processed by the GPU.

We can perform static compilation using the FXC shader compiler (`fxc.exe`) to compile our shader program once and thus eliminating the need to compile it again. The FXC HLSL compiler is invoked with its executable name followed by one or more options, the shader model profile label and the filename. For example, to compile a shader program saved in the file `shader.fx`, we can do a release build for shader model 4.0 as follows:

```
fxc /T fx_4_0 /Fo shader.fxo shader.fx
```

In this example `fx_4_0` specifies the target profile as a shader model 4.0 effect (shader model 2.0 effects are set using the `fx_2_0` profile). An effect shader can contain a combination of pixel, vertex and geometry shaders. Alternatively we could have specified the shader type as a vertex shader, pixel shader or texture shader (`tx_1_0`). These HLSL shader profiles are used to compile a shader to a specific shader model, thus ensuring hardware compatibility by limiting the supported shader model feature set. Possible Direct3D 10 vertex shader profiles include `vs_1_1`, `vs_2_0`, `vs_2_a`, `vs_2_sw`, `vs_3_0`, `vs_3_sw` and `vs_4_0` with pixel shader profiles ranging from `ps_2_0`, `ps_2_a`, `ps_2_b`, `ps_2_sw`, `ps_3_0` and `ps_3_sw` to `ps_4_0`. The `/T` switch

option specifies the HLSL profile to compile against. The `D3D10GetVertexShaderProfile`, `D3D10GetPixelShaderProfile` and `D3D10GetGeometryShaderProfile` shader functions can be called to determine the best profile suited for a given device to compile against. These functions all take a pointer to an `ID3D10Device` interface device, returning either the best vertex shader profile, pixel shader profile or geometry shader profile depending on the function called. Shader functions can be used after including the `D3D10Shader.h` header file. The next switch option, `/Fo`, is used to set the output object file name used to store the compiled shader effect.

We can alternatively compile a shader using debug mode. Debug mode is similar to that found in Visual Studio, allowing the generation of debug information and additional processing data that can be used to narrow down errors and possible bottleneck areas. We can compile the shader program saved in the `shader.fx` file using debug mode in the following manner:

```
fxc /Zi /Od /T fx_4_0 /Fo shader.fx shader.fx
```

The `/Zi` switch option enables debugging information with the `/Od` switch disabling any code-based optimisations that would normally be performed by the compiler.

B.3.2 Initialising the High Level Shader Language

This section focuses on the initialisation of the High Level Shader Language so that a Direct3D application program can bind the shader program to the appropriate pipeline stage. The steps of this initialisation process are as follows:

- 1) Compilation of the shader to ensure that the HLSL statements are syntactically correct.
- 2) Create a vertex, pixel or geometry shader object.
- 3) Set the created shader object to bind the shader to the proper pipeline stage.

A shader program is compiled by calling the `D3D10CompileShader` shader function, declared as follows in the `D3D10Shader.h` header file:

```
HRESULT D3D10CompileShader(  
    LPCSTR pSrcData,  
    SIZE_T SrcDataLen,  
    LPCSTR pFileName,  
    CONST D3D10_SHADER_MACRO *pDefines,  
    LPD3D10INCLUDE *pInclude,
```

```
LPCSTR pFunctionName,
LPCSTR pProfile,
UINT Flags,
ID3D10Blob **ppShader,
ID3D10Blob **ppErrorMsgs
);
```

Its first parameter, `pSrcData`, takes a pointer to the string holding the shader source code. The second parameter, `SrcDataLen`, specifies the size of the `pSrcData` parameter in bytes with the next parameter, `pFileName`, the name of the shader program file. The `pDefines` parameter takes a pointer to a `D3D10_SHADER_MACRO` shader macro array. This Null-terminated array of macro definitions, enabling the application program to define tokens at runtime, is optional and can be set to 'NULL'. A `D3D10_SHADER_MACRO` macro definition can be specified in the following manner:

```
D3D10_SHADER_MACRO Macro[1] = {"ten", "10"};
```

The `D3D10_SHADER_MACRO` shader structure has two members, `Name` and `Definition`. The `Name` member holds the macro name and the `Definition` member the macro definition.

`D3D10CompileShader`'s next parameter, `pInclude`, takes a pointer to the `ID3D10Include` interface allowing the opening and closing of included files when loading an effect from memory. For example, a shader program can include a file using the `#include` directive, and by calling the `Close` or `Open` `ID3D10Include` members we can open this file for reading and subsequently close it when done. Specification of the `pInclude` parameter is optional and set to 'NULL' when the shader does not contain any `#include` directives. The next parameter, `pFunctionName`, takes a pointer to a string holding the shader entry point function name indicating the function to begin the shader execution at. The `pProfile` parameter is used for setting the shader model profile with the `Flags` parameter setting the shader compile options (possible options are listed in Table B.2). The first of the final two parameters, `ppShader`, takes a pointer to an `ID3D10Blob` interface containing the debug information and compiled shader. A *blob* is a data buffer used for the storage of vertex, adjacency and material data. Blobs also return error/debug messages and object code during the compilation of pixel, vertex and geometry shaders. The last parameter, `ppErrorMsgs`, also takes a pointer to an `ID3D10Blob` interface, this time one containing errors and warning messages generated during the compilation process.



Compile Options	Description
D3D10_SHADER_AVOID_FLOW_CONTROL	The HLSL compiler will disable flow control as far possible.
D3D10_SHADER_DEBUG	The HLSL compiler enables the generation of debug information.
D3D10_SHADER_ENABLE_BACKWARDS_COMPATIBILITY	The HLSL compiler will compile older shaders to the shader model 4.0 spec.
D3D10_SHADER_ENABLE_STRICTNESS	The HLSL compiler enables strictness on deprecated shader syntax.
D3D10_SHADER_FORCE_PS_SOFTWARE_NO_OPT	The HLSL compiler will compile a pixel shader to the next best shader profile, enabling debugging and disabling compiler optimisations.
D3D10_SHADER_FORCE_VS_SOFTWARE_NO_OPT	The HLSL compiler will compile a vertex shader to the next best shader profile, enabling debugging and disabling compiler optimisations.
D3D10_SHADER_IEEE_STRICTNESS	The HLSL compiler enables IEEE strictness – thus conforming to a pre-defined set of standards.
D3D10_SHADER_NO_PRESHADER	The HLSL compiler disables the use of preshaders – an optimisation where constant expressions are replaced with references to the GPU's registers and memory addresses.
D3D10_SHADER_OPTIMIZATION_LEVEL0	The HLSL compiler enables level 0 warnings.
D3D10_SHADER_OPTIMIZATION_LEVEL1	The HLSL compiler enables level 1 warnings.
D3D10_SHADER_OPTIMIZATION_LEVEL2	The HLSL compiler enables level 2 warnings.
D3D10_SHADER_OPTIMIZATION_LEVEL3	The HLSL compiler enables level 3 warnings.
D3D10_SHADER_PACK_MATRIX_COLUMN_MAJOR	The HLSL compiler packs the matrixes in column-major order – leading to more efficiency since matrix manipulations can be performed via a series of dot-products.
D3D10_SHADER_PACK_MATRIX_ROW_MAJOR	The HLSL compiler packs the matrixes in row-major order.
D3D10_SHADER_PARTIAL_PRECISION	The compiler sets all calculations to be done with partial precision which will lead to some performance gains.
D3D10_SHADER_PREFER_FLOW_CONTROL	The HLSL compiler will enable flow control as far possible.
D3D10_SHADER_SKIP_OPTIMIZATION	The HLSL compiler will disable optimisations.

D3D10_SHADER_SKIP_VALIDATION	The HLSL compiler will disable the validation of code against common constraints and capability limits.
------------------------------	---

Table B.2 HLSL compile options.

Before calling the `D3D10CompileShader` shader function, we first have to create an `ID3D10Blob` interface:

```
ID3D10Blob * pShaderBlob;
```

We can, for instance, compile a vertex shader stored in the file `vertex_shader.vsh` as follows:

```
D3D10CompileShader(strPath, strlen(strPath),
    "vertex_shader.vsh", NULL, NULL, "EffectFunctionName", "vs_4_0",
    D3D10_SHADER_ENABLE_STRICTNESS, &pShaderBlob, NULL);
```

The shader function, `EffectFunctionName`, could have been declared in the shader program like this (taking one input parameter and returning a vertex shader structure. The declaration of shader functions, their basic, vector, texture, struct and matrix data types as well as sampler type syntax are all dealt with in the next section):

```
VS_OUTPUT EffectFunctionName (in float2 vertexPosition : POSITION)
```

A pointer to the compiled shader code is returned via the `pShaderBlob` `ID3D10Blob` interface. This pointer is used to create the vertex shader object using the `CreateVertexShader` function (for this example) or alternatively `CreatePixelShader` to create a pixel shader object or the `CreateGeometryShader` `ID3D10Device` interface function for geometry shaders. The `CreateVertexShader` function is declared in the `D3D10.h` header file as follows:

```
HRESULT CreateVertexShader(const void *pShaderBytecode, SIZE_T BytecodeLength,
    ID3D10VertexShader **ppVertexShader);
```

Its first parameter, `pShaderBytecode`, takes a pointer to the compiled shader retrieved using the `GetBufferPointer` `ID3D10Blob` interface function. The `BytecodeLength` parameter takes the size of the compiled shader determined via the `GetBufferSize` `ID3D10Blob` interface function. The final parameter, `ppVertexShader`, is the address of a pointer to an `ID3D10VertexShader` interface.

The `CreateGeometryShader` and the `CreatePixelShader` `ID3D10Device` interface functions have the same first two parameters as `CreateVertexShader`. These functions only differ in respect to the last parameter which takes a pointer to an

`ID3D10PixelShader` interface in the case of the `CreatePixelShader` function and an `ID3D10GeometryShader` interface for the `CreateGeometryShader` function.

Continuing with our vertex shader program, before calling the `CreateVertexShader` function, we specify a shader object by first declaring an `ID3D10VertexShader` interface:

```
ID3D10VertexShader **ppOurVertexShader;
```

We create the vertex shader object using the `CreateVertexShader` function (using the previously declared `ID3D10Device*` interface, `g_id3dDevice`):

```
HRESULT_ = g_id3dDevice->CreateVertexShader((DWORD*) pShaderBlob->GetBufferPointer(),  
                                           pShaderBlob->GetBufferSize(),  
                                           &ppOurVertexShader);
```

We must also remember to release the pointer to the compiled shader source:

```
pShaderBlob->Release();
```

The final step requires us to set this newly created shader object to the pipeline stage. To set the vertex shader to the device, we call the `VSSetShader` `ID3D10Device` interface function. This function takes one parameter, namely, a pointer to the `ID3D10VertexShader` vertex shader:

```
g_id3dDevice->VSSetShader(pOurVertexShader);
```

The vertex shader stage is now initialised with the compiled vertex shader code. To initialise the pixel shader stage we need to call the `PSSetShader` `ID3D10Device` interface function (using an `ID3D10PixelShader` interface as parameter). The `GSSetShader` `ID3D10Device` interface function is called for setting a geometry shader to a device (using an `ID3D10GeometryShader` interface as parameter).

B.3.3 Creating HLSL shaders

Pixel, vertex and geometry shaders each make out a different stage of the Direct3D 10 programmable pipeline. These shaders, operating on input data and sending their results to subsequent pipeline stages, are created in the form of program files that can be compiled and executed on the GPU. To recap, vertex shaders operate on vertex data with pixel shaders reading fragments (pixels) as input and geometry shaders processing primitives as input.

Vertex shaders process a vertex read as input and generates some output in the form of a transformed vertex. Vertex data are passed to the GPU via a vertex buffer. Each vertex element stored in this vertex buffer is then sent to the vertex shader for processing. For example, the following vertex shader function returns its input data as output without doing any processing on it:

```
float4 VertexShader(float4 Position : POSITION) : SV_POSITION
{
    return Position;
}
```

The vertex shader function, labelled `VertexShader`, with the return type `float4` takes a parameter, `Position`, of type `float4` as input – `float4` being a four-component HLSL vector type with each of its vector components a floating-point value. As with Cg the declaration of the input and output parameters are followed by a colon and binding semantic to further describe the data type. The input parameter is set to the `POSITION` semantic (the input vertex's clip-space coordinates) with the output value semantic set to `SV_POSITION`. Semantics using the 'sv_' prefix are referred to as system-value semantics meaning they are system generated values and can be used for both input and output data. The `SV_POSITION` semantic are, for example, processed during the rasterization stage and in this case used to notify the graphics pipeline that the output data will also be in the form of clip-space coordinates.

We can now create a pixel shader function to take the output produced by the above defined vertex shader function as input (a `float4` type coupled with the `SV_POSITION` semantic). This pixel shader then returns an output colour (red) using the `SV_TARGET` semantic that denotes the output as a render target format:

```
float4 PixelShader(float4 Position : SV_POSITION): SV_TARGET
{
    return float4(1.0f, 0.0f, 0.0f, 1.0f); //red
}
```

The next step is to specify an effect technique definition used for setting the previously defined vertex and pixel shaders. Such an effect technique, starting with the syntax, `technique10` to label it as a Direct3D 10 technique, is a set of rendering passes. Each rendering pass specifies the shader states used to render the geometry of a scene. An effect is thus a way for Direct3D to organise the states responsible for setting the stages of the graphics pipeline. The `technique10` label is followed by the name of the technique, `TechniqueName` and the name of the rendering pass, `P0`, containing the callback function(s) such as `SetPixelShader`, `SetVertexShader` or `SetGeometryShader` used to set the device state from an effect. Other states that can

be set include the blend state (`SetBlendState`) and depth-stencil state (`SetDepthStencilState`). We can create the following effect technique for the above defined vertex and fragment shaders:

```
technique10 TechniqueName
{
    pass P0
    {
        SetGeometryShader(NULL);
        SetVertexShader(CompileShader(ps_4_0, VertexShader()));
        SetPixelShader(CompileShader(ps_4_0, PixelShader()));
    }
}
```

The `SetPixelShader`, `SetVertexShader` and `SetGeometryShader` functions take a compiled shader as parameter, setting it to the appropriate render state. The geometry shader is in this case set to 'NULL' because it has not yet been defined. The vertex and pixel shaders, as well as the effect technique, are stored in an effect file (using the '.fx' file extension).

Returning to our Direct3D application, all that remains is to create the effect object and technique object that will be used for performing the rendering operation. We call the `D3DX10CreateEffectFromFile` function to create an effect from the specified effect file. This D3DX function is specified as follows in the `D3DX10Effect.h` header file:

```
HRESULT D3DX10CreateEffectFromFile(
    LPCTSTR pFileName,
    CONST D3D10_SHADER_MACRO *pDefines,
    ID3D10Include *pInclude,
    LPCSTR pProfile,
    UINT HLSLFlags,
    UINT FXFlags,
    ID3D10Device *pDevice,
    ID3D10EffectPool *pEffectPool,
    ID3DX10ThreadPump *pPump,
    ID3D10Effect **ppEffect,
    ID3D10Blob **ppErrors
);
```

This function's first parameter, `pFileName`, takes a pointer to a string containing the name of the effect file. The next parameter, `pDefines`, takes a pointer to a `D3D10_SHADER_MACRO` shader macro array with the `pInclude` parameter requiring a pointer to an `ID3D10Include` include interface as previously described. The shader

profile, as a string value, is set via the `pProfile` parameter with the HLSL compilation options being set by the `HLSLFlags` parameter. The sixth parameter, `FXFlags`, allows us to set the effect compilation options and it can be set to any of the following `D3D10_EFFECT` constants: `D3D10_EFFECT_COMPILE_CHILD_EFFECT` (the '.fx' file is compiled as a child effect, thus not initialising any shared data due to all shared values being set in the effect pool), `D3D10_EFFECT_COMPILE_ALLOW_SLOW_OPS` (compiles the effect file without performance mode) or `D3D10_EFFECT_SINGLE_THREADED` (the effect thread is not synchronised with other effects in the effect pool). An *effect pool* facilitates the sharing of variables, textures and shaders between different effects. The next parameter, `pDevice`, takes a pointer to an `ID3D10Device` interface that will use the resources to create a shader. The `pEffectPool` parameter takes a pointer to an `ID3D10EffectPool` effect pool interface signifying the memory pool used for the sharing of variables and resources between effects. The next parameter, `pPump`, is a pointer to an `ID3DX10ThreadPump` thread pump interface used for the asynchronous execution of routines; we will generally set this parameter to 'NULL' so that the `D3DX10CreateEffectFromFile` function completes its operation before returning. The second last parameter, `ppEffect`, takes the address of the pointer to the `ID3D10Effect` created effect. The `ID3D10Effect` interface is responsible for managing the shaders, state objects and resources constituting the effect. The `ppErrors` parameter is set to the address of a pointer to an `ID3D10Blob` interface. This final parameter is used for storing debug and compile-time error information.

We can create the effect using this `D3DX10CreateEffectFromFile` function in the following manner:

```
ID3D10Effect* g_id3dEffect = NULL;

D3DX10CreateEffectFromFile(L"file_name.fx", NULL, NULL,
                          D3D10_SHADER_ENABLE_BACKWARDS_COMPATIBILITY,
                          0, g_id3dDevice, NULL, NULL, &g_id3dEffect, NULL);
```

Following the effect creation we must obtain the effect technique using the `GetTechniqueByName` `ID3D10Effect` interface function. This function takes a string value containing the name of the technique as parameter, returning a pointer to the `ID3D10EffectTechnique` interface:

```
ID3D10EffectTechnique* g_id3dTechnique = NULL;

g_id3dTechnique = g_id3dEffect->GetTechniqueByName("TechniqueName");
```

A useful feature of effects is the ability to define multiple passes (subsets of a technique and a render state set – for example 'P0' in the above shown technique). We can thus define multiple passes to implement multi-pass rendering. To understand multi-pass

rendering, consider the following example. Say we have a geometry object with a texture and we decide to render some three-dimensional mesh on top of it, then we can render and texture the geometry in the first pass with the second pass being responsible for rendering the mesh on top of it. By specifying each phase as a render pass we can render both passes simultaneously during the render loop. Techniques are also useful when designing a shader to run across a vast range of hardware, for example, a technique can be specified using a pixel, vertex and geometry shader for the newest DirectX 10 hardware while another can be specified to limit the implementation to only vertex and pixel shaders so that the program can run on DirectX 9 hardware.

B.3.4 Common HLSL Data Types

HLSL features all the C++ derived scalar types such as `bool`, `int`, `float`, `string`, `double`, `uint` and `half` (a 16-bit floating point type). Shader Model 4.0 features two additional types derived from the `float` type, namely, `unorm float` (a 32-bit unsigned floating point value normalised to the range [-1, 1]) and `snorm float` (a 32-bit unsigned floating point value normalised to the range [0, 1]).

HLSL also allows for the use of vector and matrix types. Vector types can contain anything from one to four components with matrix types containing up to sixteen components. Matrix types are declared using the form `scalartypeRowxColumn`, for example, a floating point matrix, `fMatrixVar`, consisting out of four rows and three columns can be declared as follows:

```
float4x3 fMatrixVar;
```

This matrix variable can be initialised in the following manner:

```
fMatrixVar = {1.5f, 5.5f, 0.1f,  
             0.4f, 0.1f, 2.7f,  
             0.3f, 2.6f, 0.2f,  
             0.9f, 0.5f, 4.2f };
```

Matrix types can also be declared using the following syntax:

```
matrix <scalar type, number of rows, number of columns> MatrixVariableName
```

We can create the same matrix as the `fMatrixVar` one defined above using this alternate syntax:

```
matrix <float, 4, 3> fMatrixVar = {1.5f, 5.5f, 0.1f,  
                                 0.4f, 0.1f, 2.7f,
```



```
0.3f, 2.6f, 0.2f,
0.9f, 0.5f, 4.2f };
```

Vector types are declared using the syntax `ScalarType VectorVariableName`, for example, a floating point vector holding four components can be declared in the following manner:

```
float4 fVectorVar = {1.5f, 1.7f, 0.5f, 1.0f};
```

There is also, as with matrix types, an alternative syntax for the declaration of vector types:

```
vector <vector type, number of components> VectorVariableName
```

We can create the same vector, `fVectorVar`, using this alternate syntax:

```
vector <float, 4> fVectorVar = {1.5f, 1.7f, 0.5f, 1.0f};
```

HLSL also allows for the definition of structures in the following manner:

```
struct structName
{
    float variable1;
    int variable2;

    float4 fVectorVar = {1.5f, 1.7f, 0.5f, 1.0f};
    matrix <float, 4, 3> fMatrixVar = {1.5f, 5.5f, 0.1f,
                                     0.4f, 0.1f, 2.7f,
                                     0.3f, 2.6f, 0.2f,
                                     0.9f, 0.5f, 4.2f };

    ...etc
}
```

The HLSL further supports a number of operators clearly inherited from the C programming language. The most commonly used ones are listed in Table B.3.

Operator Type	Operators	Usage Examples
Additive	<code>+, -</code>	<pre>int x = 5; int y = 7; int z = x - y; int k = z + y;</pre>
Multiplicative	<code>*, %, /</code>	<pre>int x = 5; int y = 7;</pre>

		<pre>int z = x * y; float k = z / y; int l = z % y;</pre>
Array Selection	[i]	<pre>int array[2] = {3,4}; array[0] = 2;</pre>
Assignment	+=, =, *=, -=, %=, /=	<pre>int x = 5; int y = 7; int z += 3;</pre>
Bitwise	~, &, , ^, <<, >>, <<=, =, >>=, &=, ^=	<pre>z>>y //shifts the bits of z right y positions (5 >> 2 equals 1)</pre>
Boolean	, &&, ?:	<pre>bool a = false; bool b = true; bool c = a && b;</pre>
Comparison	==, !=, <, >, <=, >=	<pre>if (diffuseLight <= 0) specularLight = 0;</pre>
Prefix/Postfix Incrementing/ Decrementing	++, --	<pre>int x = 0; x++; --x;</pre>
Type Cast	(scalar type)	<pre>float x = 0.5; int y; y = (int)x;</pre>
Unary	+, -, !	<pre>bool a = true; bool b = !a;</pre>

Table B.3 HLSL Operators

B.3.5 Utilising a Created HLSL Effect

After compiling and creating an effect by loading the effect file into the effects framework (using the `D3DX10CreateEffectFromFile` function), we can proceed to initialise a number of effect constants before setting the effect state. Effects that have not yet been compiled will be compiled when they are loaded into the effects framework. Effect constants and variables are first declared in the effect/shader file(s), for example:

```
int numberOfLightSources;

float3 incomingAmbientLightColour[3];
float4 incomingDiffuseLightColour[3];
float3 objectspaceLightPosition[3];

float4x4 modelviewProjection;
float4x4 worldviewProjection;
```

```
Texture2D meshTexture;
```

These variables, declared using the HLSL data types, are set by the Direct3D application. We must thus declare variables in our application that will be used to update the shader variables:

```
int numberOfLights;  
  
D3DXVECTOR3 vIncomingAmbientLightColour [3];  
D3DXVECTOR4 vIncomingDiffuseLightColour [3];  
D3DXVECTOR3 vObjectspaceLightPosition [3];  
  
D3DXMATRIX mViewProjectionMatrix;  
D3DXMATRIX mViewMatrix;
```

Before we can set the HLSL variable values using the `ID3D10EffectVariable` update methods we first have to obtain the effect variables via `ID3D10Effect` retrieval functions for each of the above defined shader variables (this operation is similar to the retrieval of technique objects):

```
ID3D10EffectScalarVariable* g_pNumberOfLightSources;  
g_pNumberOfLightSources = g_id3dEffect  
    ->GetVariableByName("numberOfLightSources")->AsScalar();  
  
ID3D10EffectVectorVariable* g_pIncomingAmbientLightColour;  
g_pIncomingAmbientLightColour = g_id3dEffect  
    ->GetVariableByName("incomingAmbientLightColour")->AsVector();  
  
ID3D10EffectVectorVariable* g_pIncomingDiffuseLightColour;  
g_pIncomingDiffuseLightColour = g_id3dEffect  
    ->GetVariableByName("incomingDiffuseLightColour")->AsVector();  
  
ID3D10EffectVectorVariable* g_pObjectspaceLightPosition;  
g_pObjectspaceLightPosition = g_id3dEffect  
    ->GetVariableByName("objectspaceLightPosition")->AsVector();  
  
ID3D10EffectMatrixVariable* g_pWorldviewProjectionMatrix;  
g_pWorldviewProjectionMatrix = g_id3dEffect  
    ->GetVariableByName("worldviewProjection")->AsMatrix();  
  
ID3D10EffectMatrixVariable* g_pModelviewProjectionMatrix;  
g_pModelviewProjectionMatrix = g_id3dEffect
```

```
->GetVariableByName("modelviewProjection")->AsMatrix();
```

```
ID3D10EffectShaderResourceVariable* g_pMeshTexture;
```

```
g_pMeshTexture = g_id3dEffect
```

```
->GetVariableByName("meshTexture")->AsShaderResource();
```

The `GetVariableByName` `ID3D10Effect` interface function takes a string value containing the name of the variable declared in the shader/effect program as parameter, returning a pointer to the `ID3D10EffectVariable` interface. The `AsVector` `ID3D10EffectVariable` interface function casts this returned `ID3D10EffectVariable` interface to an `ID3D10EffectVectorVariable` interface so that we can access the vector type. The `AsScalar` interface function casts the returned interface to an `ID3D10EffectScalarVariable` interface used for accessing a scalar variable with the `AsMatrix` function casting it to an `ID3D10EffectMatrixVariable` interface so that we can read the shader variable as a matrix type.

Other frequently used `ID3D10EffectVariable` interface casting methods include: `AsBlend` (casts to an `ID3D10EffectBlendVariable` interface used for accessing blend-state variables), `AsDepthStencil` (casts to an `ID3D10EffectDepthStencilVariable` interface used for accessing depth-stencil variables), `AsRasterizer` (casts to an `ID3D10EffectRasterizerVariable` interface used for accessing rasterizer-state variables), `AsShader` (casts to an `ID3D10EffectShaderVariable` interface used for accessing shader variables), `AsShaderResource` (casts to an `ID3D10EffectShaderResourceVariable` interface used for accessing shader-resource variables) and `AsString` (casts to an `ID3D10EffectStringVariable` interface used for accessing string variables).

We can now set the values of the shader/effect variables using the following `ID3D10EffectVariable`, `ID3D10EffectVectorVariable`, `ID3D10EffectMatrixVariable` and `ID3D10EffectScalarVariable` methods: `SetRawValue` for generic array items, `SetFloatVectorArray` for four-component vector arrays containing floating point elements, `SetBoolVectorArray` for four-component vector arrays containing Boolean elements, `SetIntVector` for four-component vectors containing integer elements, `SetIntVectorArray` for four-component vector arrays containing integer elements, `SetMatrix` for a floating-point matrix, `SetMatrixArray` for an array of floating-point matrices, `SetFloat` for normal floating-point variables and `setInt` for integer variables:

```
g_pNumberOfLightSources->SetInt(numberOfLights);
```

```
g_pIncomingAmbientLightColour->SetRawValue(vIncomingAmbientLightColour, 0,
                                             sizeof(D3DXVECTOR3) * 3);
```

```
g_pIncomingDiffuseLightColour->SetFloatVectorArray((float*)vIncomingDiffuseLightColour,
                                                    0, 3);
```

```
g_pObjectspaceLightPosition->SetFloatVectorArray((float*)vObjectspaceLightPosition,
                                                    0, 3);
```

```
g_pWorldviewProjectionMatrix->SetMatrix((float*)&mWorldviewProjectionMatrix));
```

```
g_pModelviewProjectionMatrix->SetMatrix((float*)&mModelviewProjectionMatrix));
```

The `SetInt ID3D10EffectScalarVariable` function takes a pointer to an integer variable as parameter. The `SetRawValue ID3D10EffectVariable` function has three parameters, the first taking a pointer to the variable being set, the second specifying the offset in bytes from the beginning of the input data being set and the third the number of bytes to set from the offset value. The `ID3D10EffectVectorVariable SetFloatVectorArray` method also takes three parameters as input, namely, a pointer to the first element of a vector array, the vector offset from the start of the array to the first vector that is to be set and the number of array elements, in that order. The `SetMatrix ID3D10EffectMatrixVariable` interface function sets a floating-point matrix and is passed a pointer to the first element of a matrix as parameter.

That is it, the values declared in the shader program are now set and can be changed during each rendering pass. The final step is to set the effect state within the device itself. This is done by invoking the effect state from within the render loop by selecting a technique and subsequently setting the state for each of the passes:

We start by calling the `GetDesc ID3D10EffectTechnique` function on the previously defined technique object which is used for storing the returned `D3D10_TECHNIQUE_DESC` structure, i.e. the structure describing the technique:

```
ID3D10EffectTechnique* g_pd3d10EffectTechnique = NULL;

/* obtain the D3D10_TECHNIQUE_DESC effect-variable description */
D3D10_TECHNIQUE_DESC technique;
g_pd3d10EffectTechnique->GetDesc(&technique);
```

The `GetPassByIndex ID3D10EffectTechnique` interface method is now called to acquire an effect pass object representing the first pass of the technique:

```
/* apply the effect state by looping over the number of technique passes */
for(int i = 0; i < technique.Passes; ++i)
{
```



```
g_pd3d10EffectTechnique->GetPassByIndex(i)->Apply(0);  
...etc  
}
```

Lighting and Reflection

C.1 Lighting

Before considering shadows, it is 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 C.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 C.1 (b) shows this same scene illuminated by a properly defined light source.

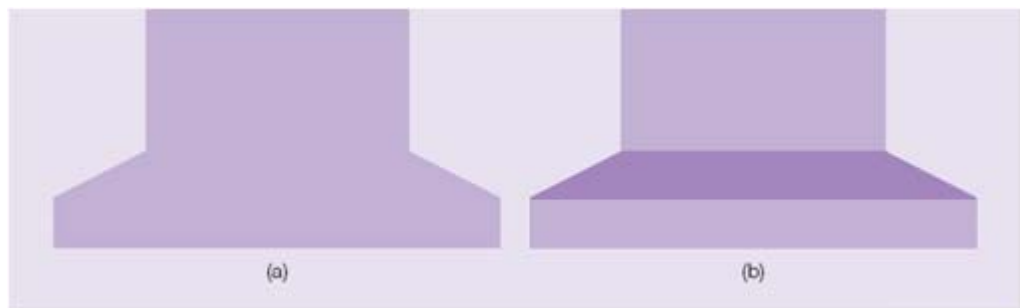


Figure C.1 (a) Two rendered rectangles, the one representing a floor, the other a facing wall. (b) The same rectangles with lighting enabled.

This lack of depth is the result of uniform lighting, i.e. the equal illumination of all surfaces. Figure C.2 (a) shows a uniformly lit sphere and Figure C.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.

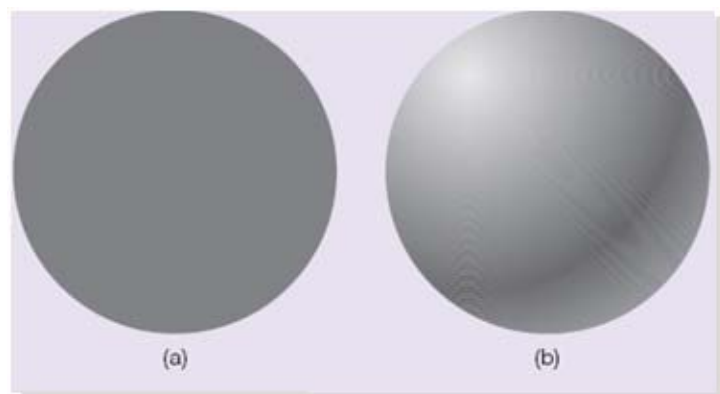


Figure C.2 (a) A uniformly lit sphere and (b) a properly lit and shaded sphere.

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 C.3.

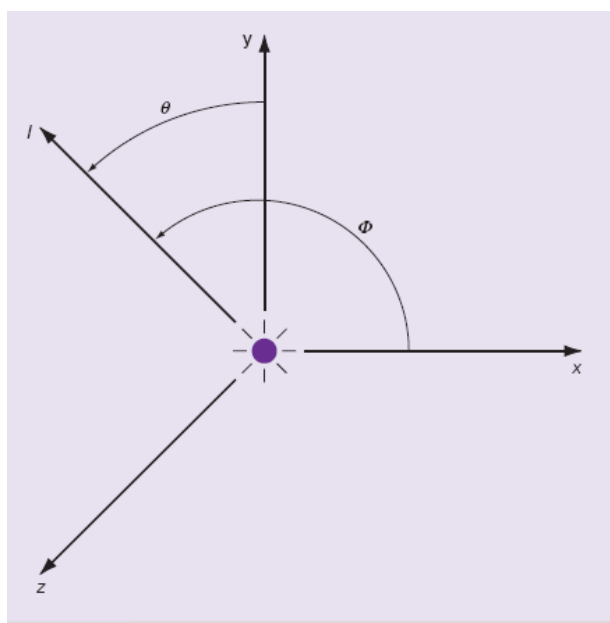


Figure C.3 A basic light source characterised via six elements.

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 C.4 shows two distinct illumination functions for a pair of points located on the surface of a light source.

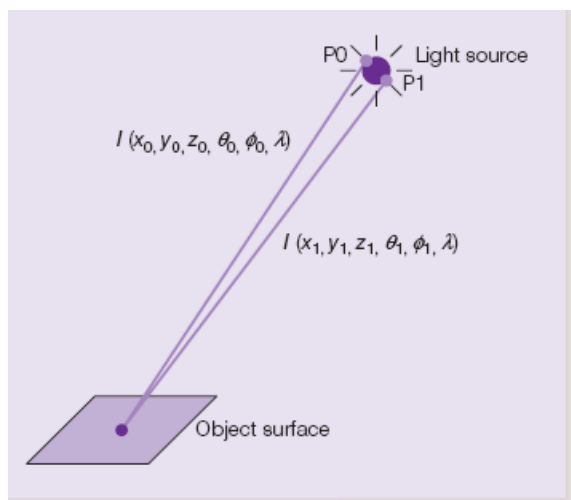


Figure C.4 Two distinct illumination functions for a single light source.

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 it is 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.

C.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 (Newman and Sproull, 1973). 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 C.5 illustrates the effect of a point light illuminating a surface.

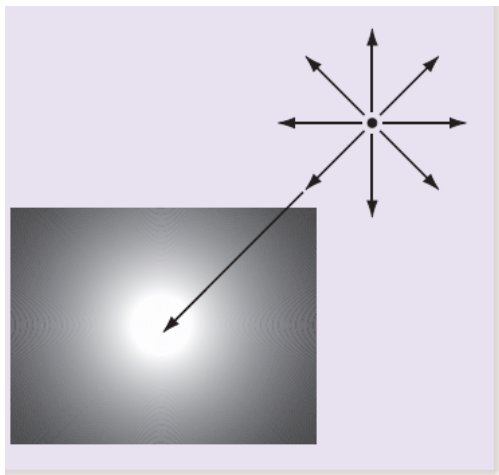


Figure C.5 Point light illumination.

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:

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

C.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 C.6 illustrates this diminishing property.

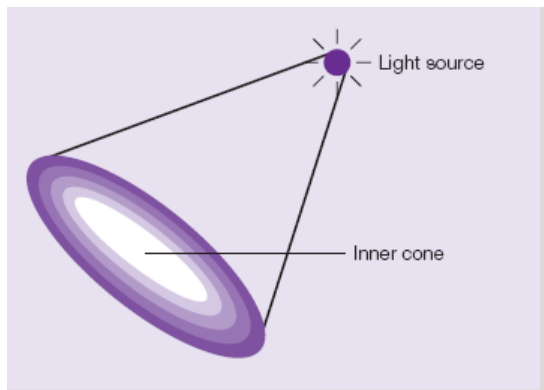


Figure C.6 Spotlight falloff.

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:

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

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 C.7):

$$\cos \theta = (\text{spotlightDirectionVector}) \bullet (\text{vectorToSurface}).$$

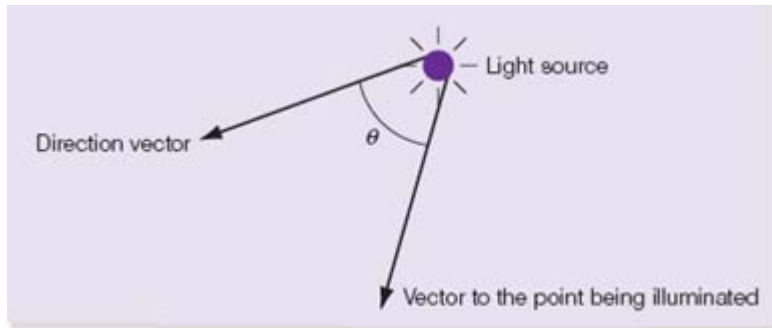


Figure C.7 The relationship between the direction vector and the vector to the point being illuminated.

C.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):

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

C.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 C.8 illustrates a parallel light source.

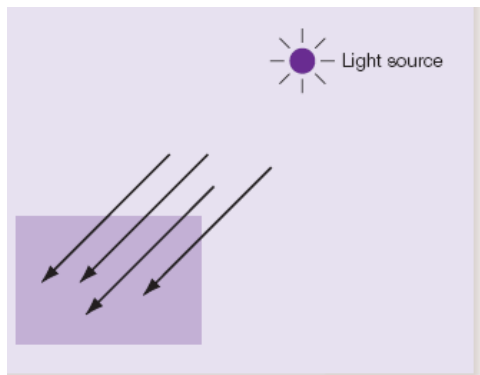


Figure C.8 A parallel light.

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.

C.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:

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

C.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 (Schlick, 1993).

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 (Sillion and Puech, 1989):

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 (Walter et al, 1997):

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).

C.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 (Warn, 1983). 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 C.9 illustrates this concept.

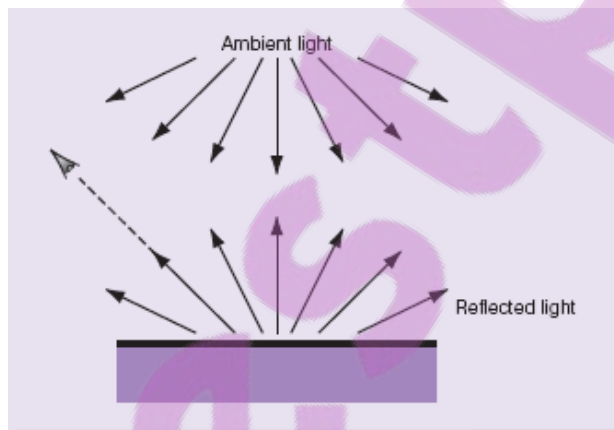


Figure C.9 Ambient reflection

The problem with ambient reflection is that illuminated objects appear rather flat and unshaded; Figures C.1 (a) and C.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 – R_{Ar} , R_{Ag} and R_{Ab} , representing the red, green, and blue ambient reflection coefficients, respectively:

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

C.2.2 Specular Reflection Model

Specular reflection occurs whenever light, from a single incoming direction, is reflected at a single outgoing direction (Torrance and Sparrow, 1967). 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 C.10.

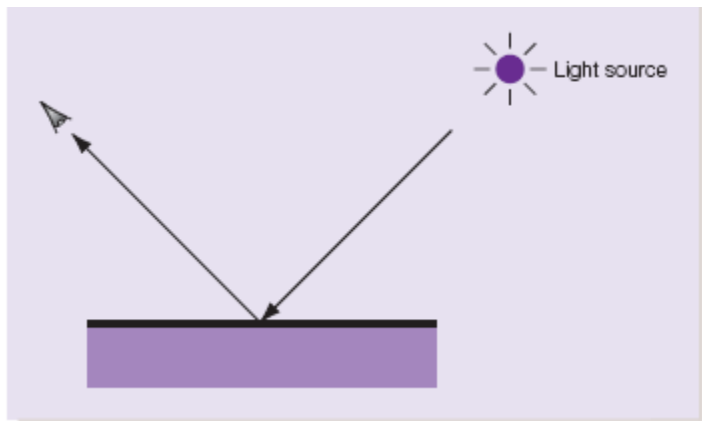


Figure C.10 Specular reflection

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 C.11 shows several spheres with specular highlights.

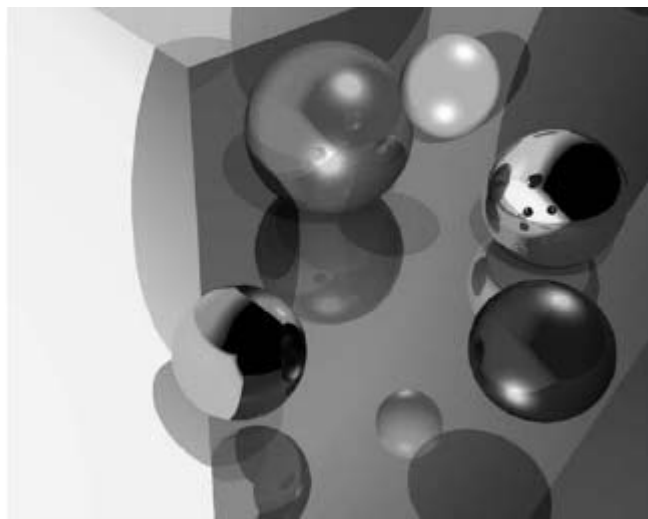


Figure C.11 Examples of specular highlights

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 C.12 shows the vectors used in the calculation of this specular term.

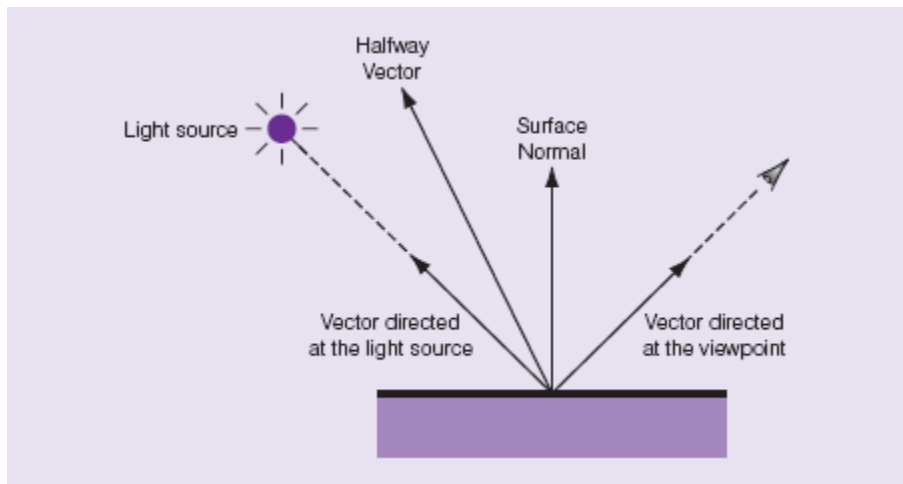


Figure C.12 Vectors used in the calculation of the specular term

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 R_s , the fraction of the incoming specular light being reflected:

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

C.2.3 Diffuse Reflection Model

Diffuse reflections occur when incoming light is reflected in arbitrary directions (Goral et al, 1984). 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 (Hall, 1989). The equation for calculating diffuse lighting is:

$$\begin{aligned} \text{Diffuse lighting term} &= \text{material's diffuse color} \\ &\times \text{color of incoming diffuse light} \\ &\times \max(\text{normalized surface normal} \\ &\quad \cdot \text{normalized vector towards light}, 0) \end{aligned}$$

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 C.13 shows a diffuse surface with the normalized surface normal and normalized vector pointing at the light source.

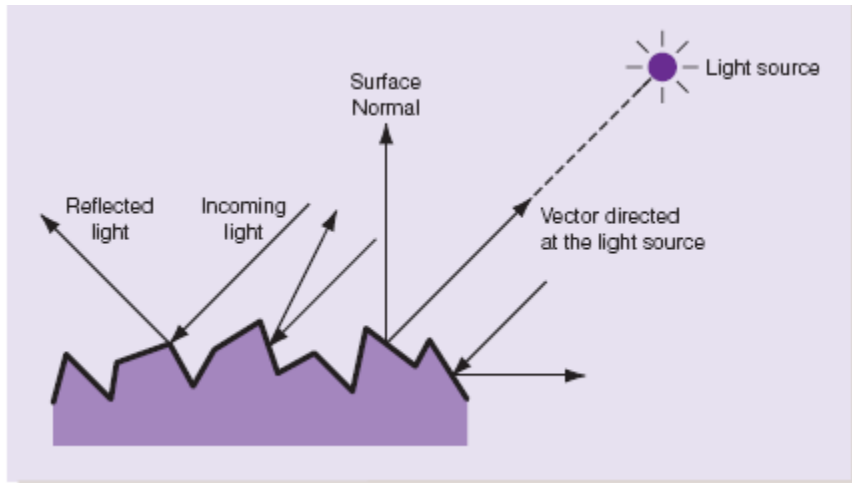


Figure C.13 Diffuse reflections

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:

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

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 C.14 illustrates Lambert's cosine law.

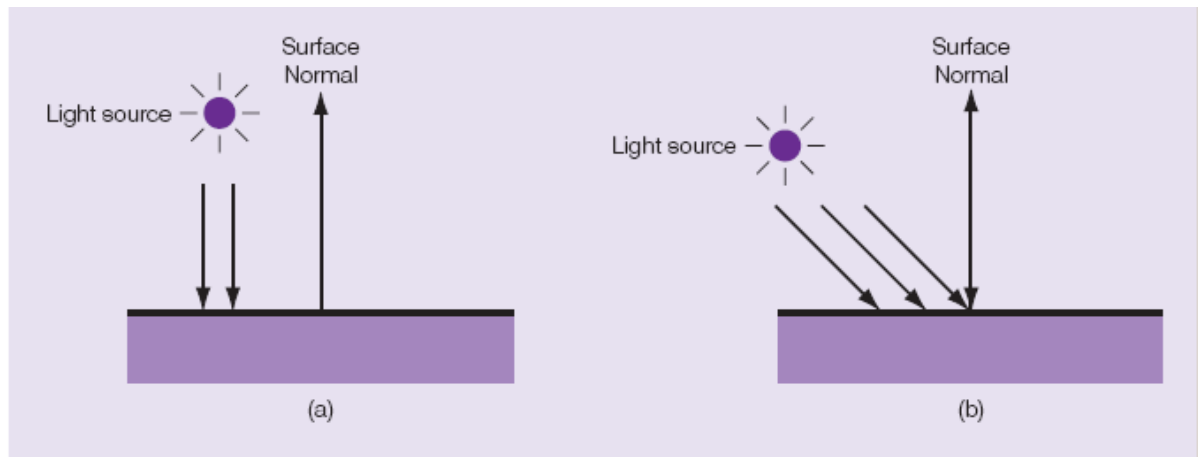


Figure C.14 A perfect diffuse surface being illuminated by (a) a light source positioned perpendicular to the surface and (b) a light source positioned at a 135 degree angle

C.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 C.15. We can consequently write the combination of these lighting terms as:

$$I = I_a + I_d + I_s.$$



Figure C.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 C.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).

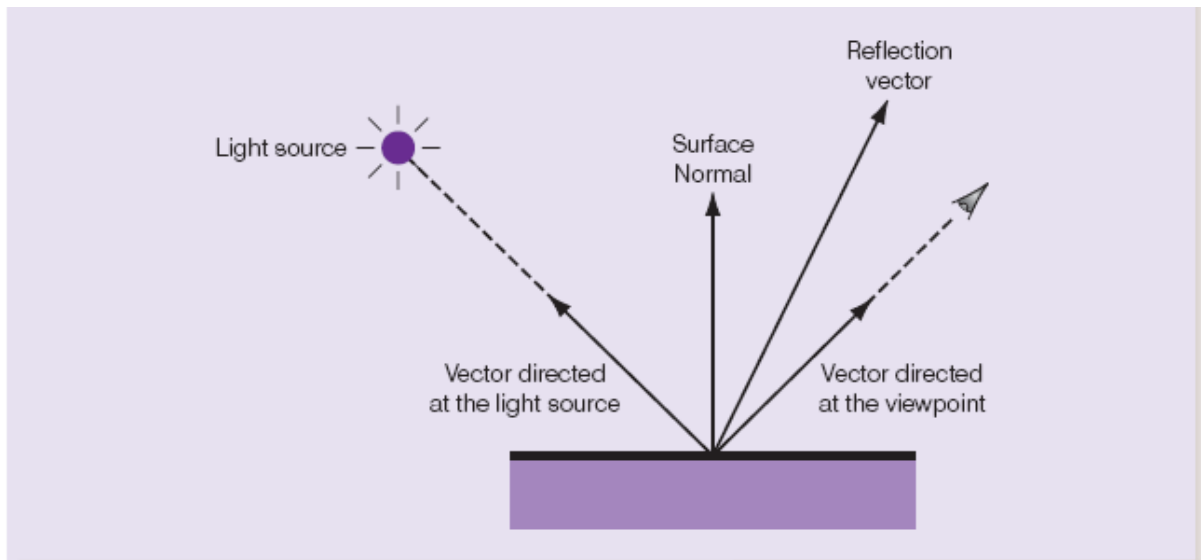


Figure C.16 Vectors used in the calculation of the Phong reflection model

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

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

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, α the shininess coefficient and i_s 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.

Real-time Shadow Generation

D.1 Introduction

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 D.1 illustrates shadow generation, specifically the implementation of stencil shadow volumes – a popular shadow rendering technique.

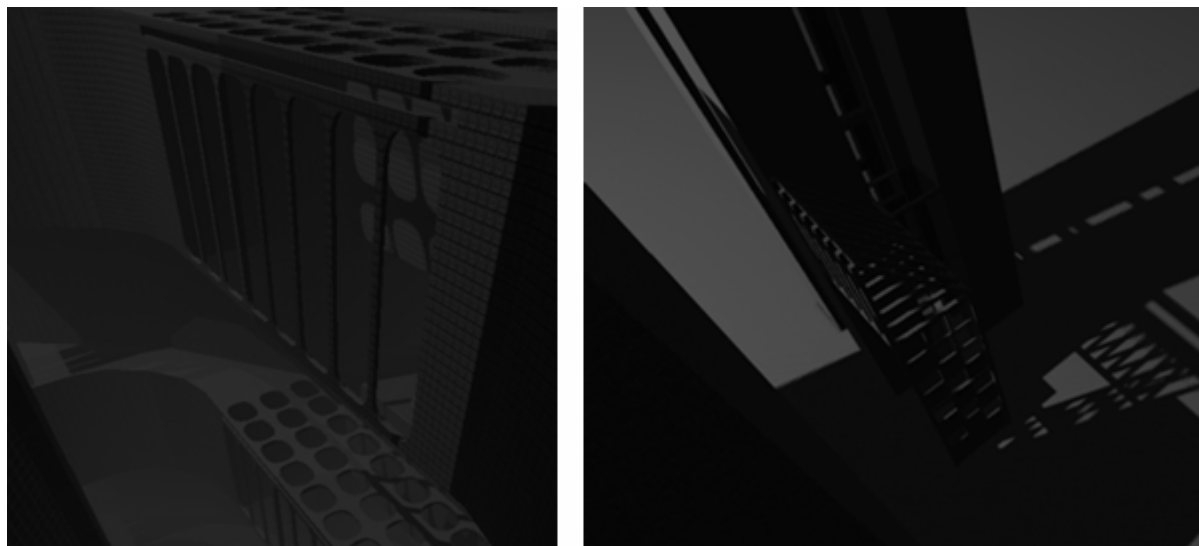


Figure D.1 Example of stencil shadowing – note the darkening of overlapping shadows.

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 D.2 illustrates a shadow's umbra and penumbra.

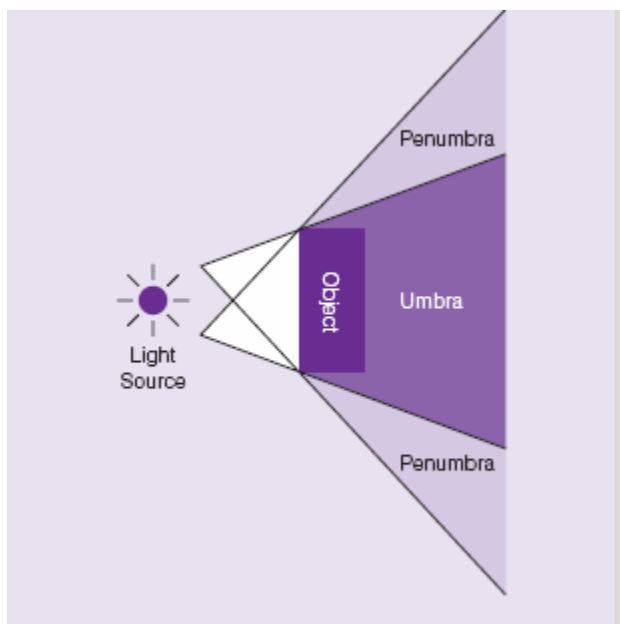


Figure D.2 A soft shadow with related umbra and penumbra.

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 D.3 shows the difference between shadows produced by point and ambient light sources.

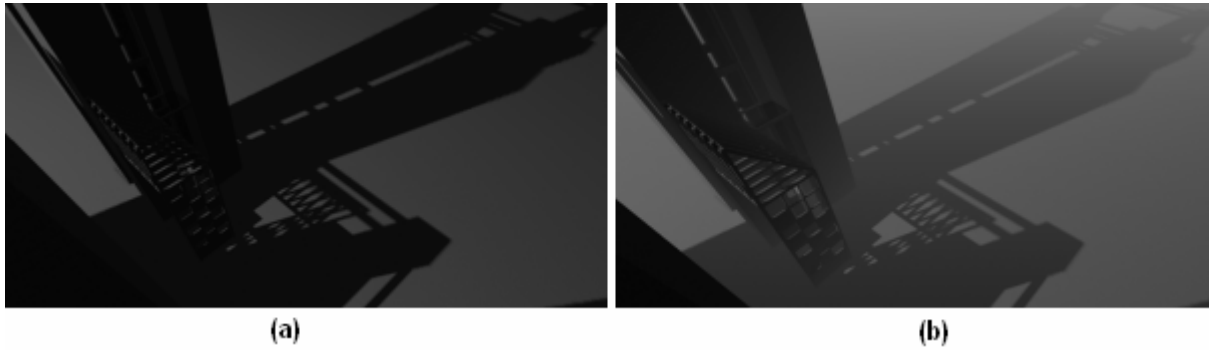


Figure D.3 (a) Hard-edged shadow produced by a point light source. (b) Soft-edged shadow produced by an ambient light source.

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.

D.2 Shadow Rendering Algorithms

D.2.1 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 (Lane et al, 1980). 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 (Bresenham, 1987). 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 D.4 illustrates the above described process.

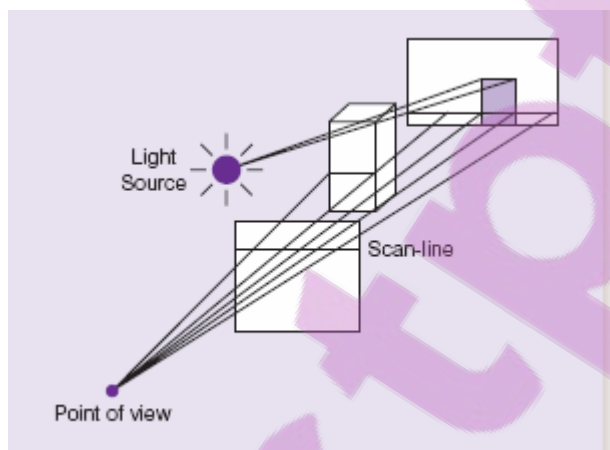


Figure D.4 Scan-line polygon projection.

D.2.2 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 D.5 illustrates the projection of a shadow polygon (onto the xy -plane) with the light source located at the centre of projection.

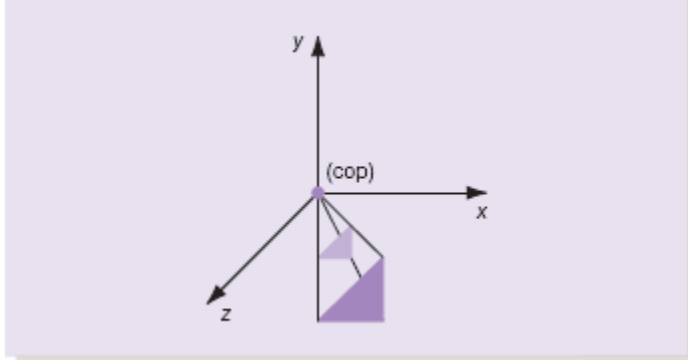


Figure D.5 Shadow polygon with a point light source at the centre of projection.

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:

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

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

- 1) Define and initialise the shadow projection matrix \mathbf{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.

D.2.3 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 (Shade et al, 1998). 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 D.6 shows a 3-D object and its resulting shadow map.

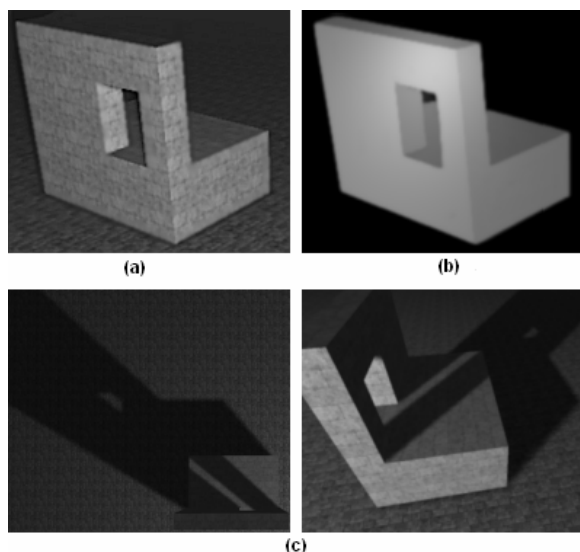


Figure D.6 (a) Object as seen from the light's point of view (b) Object's depth map from the light's point of view (c) Shadow polygon rendered via the horizontal projection of the depth map.

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 (Fernando et al, 2001). 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 (Lauritzen, 2006). 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.

D.2.4 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 D.7 shows the silhouette edge of a cube with Figure D.8 illustrating the capping of a semi-infinite shadow volume.

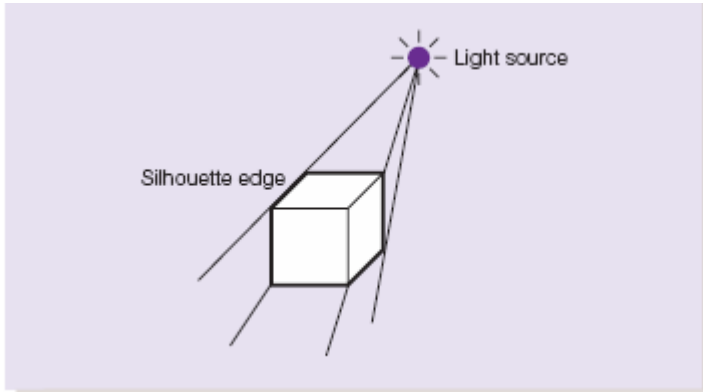


Figure D.7 A simple silhouette edge.

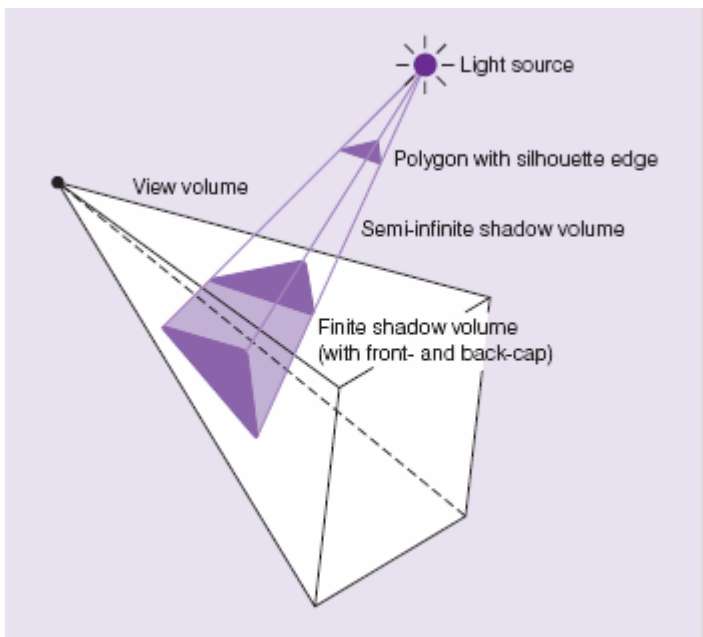


Figure D.8 Construction of finite shadow volume.

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 is 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.

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 D.9 illustrates the above described process:

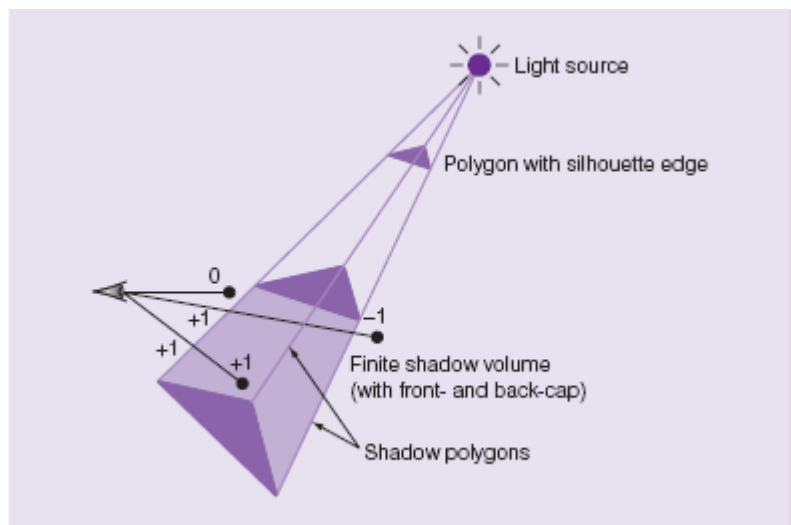


Figure D.9 Testing whether a fragment is in shadow.

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.

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 et 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.

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 D.10):

- 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.

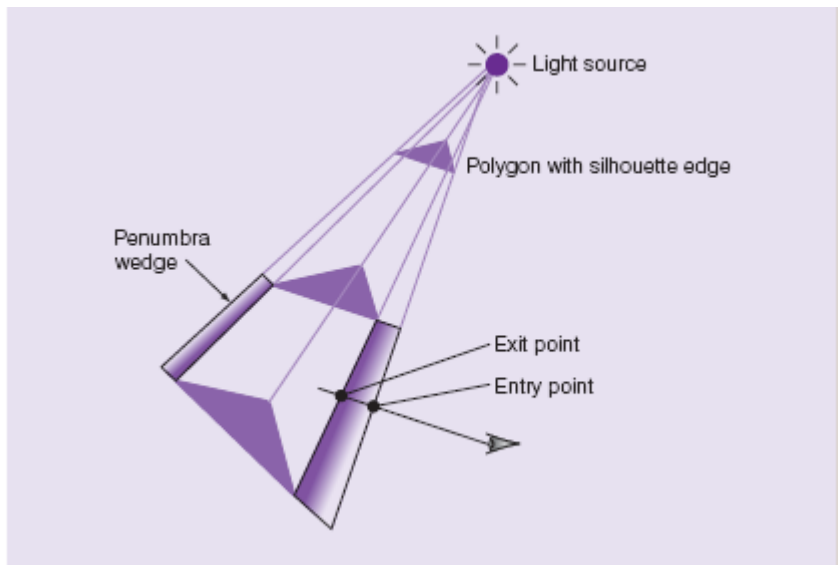


Figure D.10 Locating a point within the penumbra region.

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.

Physics

Simulating Newtonian physics through the use of quantities such as mass, acceleration, velocity, friction, momentum, force, etc allows for the prediction of object behaviour under certain conditions (Halliday et al, 2007). For example, through physics modelling we can simulate the expected behaviour of several stacked barrels falling over or even an explosion ripping through a bunker complex.

Physics modelling is generally implemented as part of a physics engine. Physics engines are classified into two classes: real-time engines such as the Havok physics engine and high-precision physics engines such as those used by scientists. *Real-time physics engines* “approximate” physics modelling to balance computational accuracy with the speed of the simulation (as the case with our quality scaling). *Scientific physics engines* are employed by organisations like NASA and universities for various simulations, for example, Figure E.1 shows the computational fluid dynamics model used for simulating the air flow around a space shuttle during atmospheric re-entry.

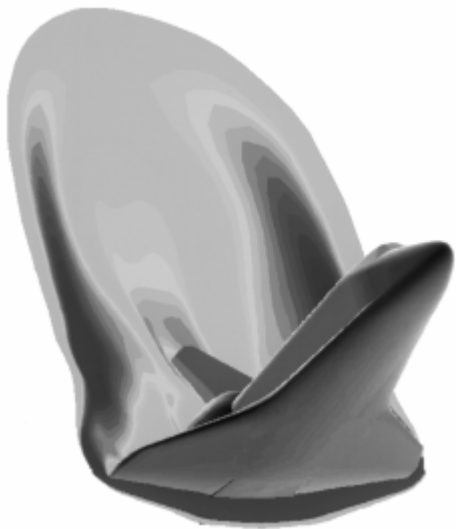


Figure E.1 Simulated air flow around a space shuttle during atmospheric re-entry.

The shown computational fluid dynamics model requires an incredible amount of processing power to simulate (Belleman et al, 2008). This is mostly due to the use of numerical methods and advanced algorithms when analysing the flow of particles – each particle is assigned a force vector which are then combined across the entire region to illustrate the resulting particle flow (Reeves, 1983).

When adding Newtonian physics to a game we must always keep processing constraints in mind. Our biggest problem is not performing the physics calculations but

dealing with a fluctuating frame rate and rounding errors that can result in unrealistic motion (Witken and Heckbert, 1994). On the other hand, increasing data precision will solve the problem of rounding errors (Reeves and Blau, 1985) but with a significant impact on CPU and/or GPU usage.

We will now model Newtonian physics by looking at the conservation and transfer of momentum as well as the modelling of gravitational pull, trajectories, friction and object collision.

E.1 Linear Momentum

Action-oriented games without collisions would simply not work. Whether it is a projectile fired from a weapon striking a monster, a car skidding across the Daytona Speedway or the player activating a switch; without the ability to simulate one object striking another we would simply not “have game”.

At the core of collision simulation is the conservation and transfer of momentum (Moore and Wilhelms, 1988). The *conservation of momentum* is described as a rule of nature stating that if we have a closed system of objects, without any external interaction, then the total momentum of this system will remain constant. This rule links back to Newton’s first law of motion, that is, a body in motion will remain in motion unless a net force is exerted upon it. Building on this; Newton’s third law of motion states that for every action there is an equal and opposite reaction – a law that can be proven by considering the conservation of momentum.

To understand conservation of momentum, consider a game of squash in a perfect world where no energy is lost when the ball hits the squash court’s wall (in the real world energy will be released in the form of sound, heat and deformation the moment the ball hits the wall, thus resulting in a slower velocity (and less momentum) after the collision than before. However, in a perfect world we don’t consider loss in momentum and the velocity of the ball remains the same after the collision than as before.

The *transfer of momentum* describes the situation where a collision occurs and momentum is transferred from the one object to the other. Thus, the lost of momentum at the one side must equal the momentum gained at the other (assuming conservation of kinetic energy as well as momentum before and after the collision). This concept is described mathematically as follows:

$$\Delta p_{object1} = -\Delta p_{object2}, \text{ where } \Delta p \text{ is the change in momentum of each object.}$$

A well known example demonstrating the conservation and transfer of momentum is Newton's cradle – a device consisting of five (or more) pendulums neighbouring one another. Figure E.2 shows Newton's cradle, when the midair pendulum is released, it will collide with the left-most static pendulum. On impact, energy is transferred from one pendulum to the other until the right-most pendulum is pushed outwards by the transferred force. The motion will eventually cease due to a continuous energy loss (mostly released as sound energy i.e. "clacking" sounds).

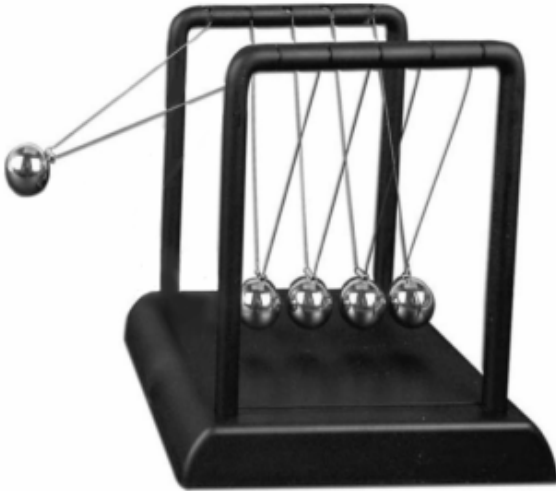
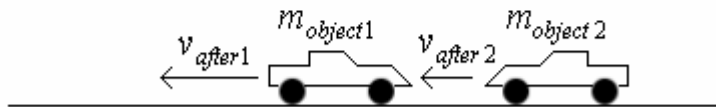


Figure E.2 Newton's cradle used for demonstrating the conservation/transfer of momentum and energy.

To fully understand perfect collisions and the conservation of momentum, consider the two objects shown in Figure E.3.



(a) Before object collision



(b) After object collision

Figure E.3 Collision and the transfer of momentum.

Both objects have a mass ($m_{object1}$ and $m_{object2}$) and initial velocity ($v_{initial1}$ and $v_{initial2}$). After collision each will have a new velocity – two unknown values at this stage (v_{after1}

and v_{after2}). Using these variables we can now describe the conservation of momentum mathematically using the following equation:

$$m_{object1} \times v_{initial1} + m_{object2} \times v_{initial2} = m_{object1} \times v_{after1} + m_{object2} \times v_{after2}$$

One problem with this equation is that we normally wish to calculate each object's vector velocity after the collision, something which is impossible because we'll always end up with two unknowns. For example, say object 1 has a mass of 250kg and an initial velocity of 1200m/s while object 2 has a mass of 300kg and an initial velocity of 2400m/s, then by substituting these values in the above equation, we get:

$$250kg \times 1200m/s + 300kg \times 2400m/s = 250kg \times v_{after1} + 300kg \times v_{after2}$$

The only logical approach is to combine this equation with something we already know, in this case the conservation of energy, specifically the *conservation of kinetic energy*.

Kinetic energy is energy stored in a moving object, or more specifically, the mechanical work needed to accelerate this object from rest to its current state. *Mechanical work* is the total amount of energy transferred to an object through the application of force. The simplest way of calculating work, measured in joule (J), is to use the following formula:

$W = Fd$, where F is the force exerted on the object and d the distance travelled by the object.

This formula can also be written as:

$$W = \frac{1}{2}mv^2, \text{ with } m \text{ the mass of the object and } v \text{ its velocity.}$$

Applying external work to an object causes a change in its kinetic energy. For example, say an object has an initial kinetic energy of $E_{k_initial}$ and some force is applied to it resulting in a new kinetic energy, E_{k_final} , then we can represent the relation between work and kinetic energy as follows:

$$\begin{aligned} W &= \Delta E_k \\ &= E_{k_final} - E_{k_initial} \end{aligned}$$

Kinetic energy (E_k) is the ability to do work and can easily be calculated using the following equation:

$E_k = \frac{1}{2}mv^2$, with m the mass of the object in kilograms and v its velocity in meters per second.

Kinetic energy, akin to work, is measured in Joules (J), with one Joule being equal to one kilogram-meter squared per second squared (kgm^2/s^2). This energy remains constant before and after a collision – a condition described as the *conservation of kinetic energy*. In the real world energy will of course be lost in the form of sound, heat and deformation; however, this is only something that will be considered for the implementation of a scientific physics engine. Using this conservation property we can now describe the total kinetic energy before and after a collision via the following equation:

$$\frac{1}{2}m_{\text{object1}} \times v_{\text{initial1}}^2 + \frac{1}{2}m_{\text{object2}} \times v_{\text{initial2}}^2 = \frac{1}{2}m_{\text{object1}} \times v_{\text{after1}}^2 + \frac{1}{2}m_{\text{object2}} \times v_{\text{after2}}^2$$

We can now use this equation in combination with the previous listed one describing the conservation of momentum to solve the given example's two unknown velocities following the collision:

$$250\text{kg} \times 1200\text{m/s} + 300\text{kg} \times 2400\text{m/s} = 250\text{kg} \times v_{\text{after1}} + 300\text{kg} \times v_{\text{after2}}$$

$$\frac{1}{2}(250\text{kg})(1200\text{m/s})^2 + \frac{1}{2}(300\text{kg})(2400\text{m/s})^2 = \frac{1}{2}(250\text{kg})v_{\text{after1}}^2 + \frac{1}{2}(300\text{kg})v_{\text{after2}}^2$$

The simplest approach would be to write v_{after1} in terms of v_{after2} for the second equation, substituting it into the first equation and solving v_{after2} .

Our treatment of particles extends this discussion by looking at the simulation of bouncing objects and inter-object collision detection and response.

E.2 Gravitational Pull

When looking at any early 1990s side-scrolling game, such as *Super Mario World* or *Commander Keen*, one can quickly see the effect of gravity on the player. For example, jumping vertically into the air is quickly followed by the game character returning to its previous position. This is an early example of gravity in games with modern games modelling gravity much more closely.

Gravity is the natural phenomenon where objects attract each other due to each object being surrounded by a gravitational field. This field, interpreted as an attractive power, exerts a pulling force on all surrounding objects, as shown in Figure E.4.

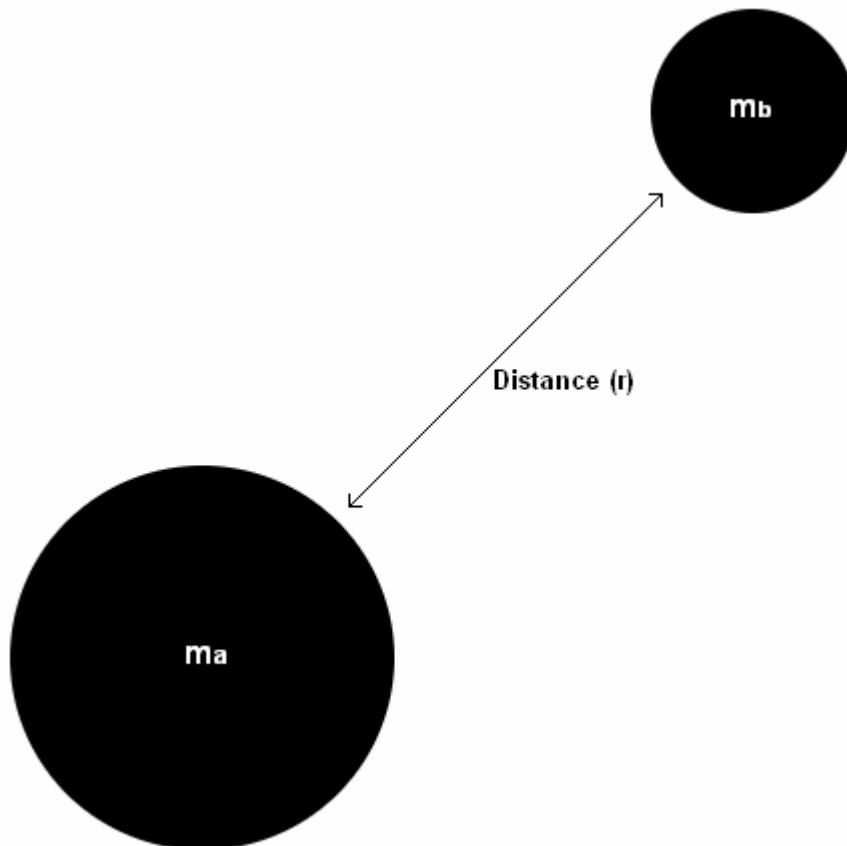


Figure E.4 The gravitational pull between two objects of mass m_a and m_b , respectively.

Each of the two objects shown in Figure E.5 will experience the effect of gravity, with the exact gravitational force between the two objects given by the following equation:

$F = \frac{G \times m_a \times m_b}{r^2}$, where G is the universal gravitational constant (equal to $6.67 \times 10^{-11} \text{Nm}^2/\text{kg}^2$), m_a the mass of the one object and m_b the mass of the other with r the distance in meters between the two objects.

Simulating gravity in games does not generally require advanced calculations that involve the universal gravitational constant or the exact mass of an object. For example, when modelling gravity for an object being dropped to the ground, we can start with the assumption that the acceleration of this object will be 9.8m/s^2 regardless of its mass (standard acceleration due to the earth's gravitational field). We can now define the velocity and position of this object as follows:

$$V_{new} = V_{old} + (9.8)t$$

$$\begin{aligned} Pos_{new} &= Pos_{old} + (v_{old} \times t) + \left(\frac{1}{2} \times a \times t^2\right), \\ &= Pos_{old} + (v_{old} \times t) + \left(\frac{1}{2} \times 9.8m/s^2 \times t^2\right) \end{aligned}$$

Now, let's assume a crate is dropped at an initial velocity of 0 m/s from a position located at coordinates (0, 17, 0) as shown in Figure E.5.

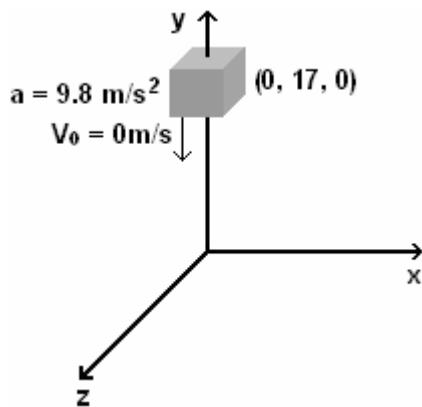


Figure E.5 Gravitational attraction of an object towards the zx-plane.

Substituting these values into the above given equations yield the following equations (assuming the coordinate $y = 17$ equates to a virtual height of 17 meters):

$$\begin{aligned} V_{new} &= 0m/s + (9.8)t \\ &= 9.8t \end{aligned}$$

$$\begin{aligned} Pos_{new} &= 17m + (0m/s \times t) + \left(\frac{1}{2} \times 9.8m/s^2 \times t^2\right), \\ &= 17m + \left(\frac{1}{2} \times 9.8m/s^2 \times t^2\right) \end{aligned}$$

We can now implement these equations in the following manner – thus simulating gravity:

```
/* initialise the object's initial position */
float objectXPos = 0;
float objectYPos = 17;
float objectZPos = 0;
```

```
/* set the object's initial velocity */
float objectXVelocity = 0;
float objectYVelocity = 0;
float objectZVelocity = 0;

/* initialise the object's rate of fall - hence its gravity */
float worldGravityConstant = 1.5f;

/* use a loop to update the object's position and velocity until the zx-plane is
   reached */
while(objectYPos > 0)
{
    /* increase the velocity as the object falls */
    objectYVelocity = objectYVelocity + worldGravityConstant;

    /* calculate the object's new position */
    objectYPos = objectYPos + objectYVelocity;
}
}
```

This object will only fall in a straight vertical line, by incrementally adjusting its x-coordinate in the loop, for example, we can simulate a curved falling trajectory as shown in Figure E.6.

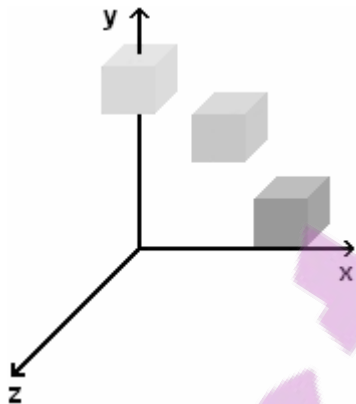


Figure E.6 Gravitational attraction of an object thrown in the x-direction.

We can now modify the above listed code snippet to simulate a curved falling trajectory as follows:

```
/* initialise the object's initial position */
float objectXPos = 0;
float objectYPos = 17;
float objectZPos = 0;
```

```

/* set the object's initial velocity */
float objectXVelocity = 0;
float objectYVelocity = 0;
float objectZVelocity = 0;

/* initialise the object's rate of fall - hence its gravity */
float worldGravityConstant = 1.5f;

/* use a loop to update the object's position and velocity until the zx-plane is
   reached */
while(objectYPos > 0)
{
    /* increase the velocity as the object falls */
    objectYVelocity = objectYVelocity + worldGravityConstant;

    /* calculate the object's new y-position */
    objectYPos = objectYPos + objectYVelocity;

    /* calculate the object's new x-position by adding a constant x velocity */
    objectXPos = objectXPos + 3;
}

```

E.3 Trajectory Paths

Without accurate projectile simulation, we would not be able to model bomb drops from aeroplanes, a kick off in a football game or the trajectory of a baseball after being hit by a batter. Figure E.7 shows the trajectory path of a ball being kicked in the positive x-direction.

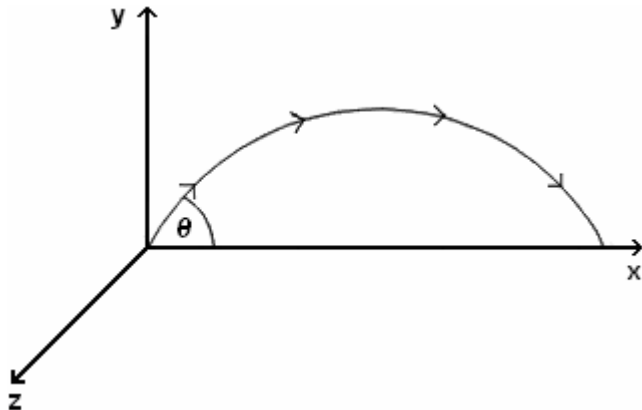


Figure E.7 The trajectory path of a ball being kicked in the positive x-direction.

Trajectory can be described as the path or course travelled by an object. Calculating this path often requires the consideration of gravitational forces, aerodynamic factors, wind shear, etc. For most game-based implementations we'll assume uniform gravity while negating wind and other aerodynamic factors. For example, to model the trajectory path shown in Figure E.7 we can define the ball's initial velocity in terms of an x- and y-component as follows (with θ the inclination angle):

$$V_x = V_{initial} \times \cos \theta$$

$$V_y = V_{initial} \times \sin \theta$$

We can also assume that V_y will equal "0" at the apex of the arch (the maximum height reached by the projectile).

Modelling a trajectory path involves applying a constant velocity along the x-axis (in the case of the above shown path) as well as the effect of gravity in the direction of the negative y-axis. We also factor in air resistance without needlessly complicating our simulation. The following code sample simulates a trajectory path as illustrated in Figure E.7:

```

/* initialise the object's initial position */
float objectXPos = 0;
float objectYPos = 0;
float objectZPos = 0;

/* set the object's initial velocity */
float objectXVelocity = 0;
float objectYVelocity = 0;
float objectZVelocity = 0;

/* initialise the object's rate of fall - hence its gravity */
float worldGravityConstant = 1.5f;

/* set the inclination angle to 45 degrees in radians */
float initialAngle = 0.79

/* set the air resistance that will be factored in to simulate the deceleration of the
   projectile */
float airResistance = 0.01f

/* calculate the velocity's x- and y-component */
objectXVelocity = objectXVelocity*cos(initialAngle);
objectYVelocity = objectYVelocity*sin(initialAngle);

```

```

/* use a loop to update the object's position and velocity until the zx-plane is
   reached */
while(objectYPos > 0)
{
    /* update the object's velocity */
    objectYVelocity = objectYVelocity + worldGravityConstant;
    objectXVelocity = objectXVelocity - airResistance;

    /* calculate the object's new y-position */
    objectYPos = objectYPos + objectYVelocity;

    /* calculate the object's new x-position */
    objectXPos = objectXPos + objectXVelocity;
}

```

E.4 Friction

Friction, stemming from electromagnetic forces between atomic particles, is an energy consuming force between two objects in contact. The most common form of friction is known as Coulomb friction. *Coulomb friction* is an approximation stating that the maximum force exerted by friction (F_f) is always less than or equal to the direct normal force (F_n) between two objects multiplied by the material's friction coefficient (μ):

$$F_f \leq F_n \times \mu$$

The *normal force* (shown in Figure E.8) is a force component perpendicular to the surface of contact with the coefficient of friction an empirically determined constant that varies depending on the type of material surface and whether the surface is perfectly clean, etc.

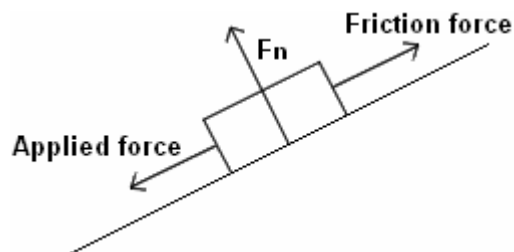


Figure E.8 The normal, friction and applied (sliding) forces exerted on an object.

Table E.1 gives some of the most common friction coefficients; also note that friction varies depending on whether an object is static or in motion.

Material	Static	In Motion (kinetic)
Aluminium on aluminium	1,05-1,35	1,4
Aluminium on steel	0,61	0,47
Copper on cast iron	1,05	0,29
Copper on steel	0,53	0,36
Glass on glass	0,9 - 1,0	0,4
Glass on nickel	0,78	0,56
Leather on wood (along the grain)	0,61	0,52
Nickel on nickel	0,7-1,1	0,53
Nylon on nylon	0,15 - 0,25	
Steel on steel (high level hardness)	0,78	0,42
Steel on steel (relative hardness)	0,74	0,57
Wood on wood (against the grain)	0,54	0,32
Wood on wood (along the grain)	0,62	0,48

Table E.1 Common coefficients of friction.

We generally calculate the force required to move a static object via the following equation:

$F_f = m \times g \times \mu_{static}$, where m is the mass of the object, g the gravitational constant ($9.8m/s^2$) and μ the material's static friction coefficient.

The object will only move once a force greater than F_f is applied to it, after which its friction coefficient normally decreases. For example, consider an aluminium object weighing 90 kilograms placed on a flat polished steel surface – we can calculate the maximum force exerted by friction as follows:

$$\begin{aligned}
 F_f &= m \times g \times \mu_{static} \\
 &= 90kg \times 9.8m/s^2 \times 0.61 \\
 &= 538.02N
 \end{aligned}$$

We will thus require a force of at least 538.03N to move this object, once it is in motion we can recalculate it frictional force using aluminium on steel's kinetic friction coefficient:

$$\begin{aligned}
 F_f &= m \times g \times \mu_{kinetic} \\
 &= 90kg \times 9.8m/s^2 \times 0.47 \\
 &= 414.54N
 \end{aligned}$$

Friction on a flat plane can be modelled just like air resistance (which is in fact a form of friction):

```
/* initialise the object's initial position */
float objectXPos = 0;
float objectYPos = 0;
float objectZPos = 0;

/* set the object's initial velocity */
float objectXVelocity = 15;
float objectYVelocity = 0;
float objectZVelocity = 0;

/* set the friction value */
float friction = 1.5f

/* use a loop to update the object's position and velocity until the object's speed
reaches zero */
while(objectXVelocity > 0)
{
    /* update the object's velocity */
    objectXVelocity = objectXVelocity - friction;

    /* calculate the object's new x-position */
    objectXPos = objectXPos + objectXVelocity;
}
}
```

E.5 Simulating Object Collisions

Let's start with a two dimensional "asteroid field" from Atari's 1979 cult-hit, Asteroids. This game, as shown in Figure E.9, is heavily dependent on object collisions such as asteroids colliding with other asteroids, alien spaceships or with the player's ship.



Figure E.9 Screenshot of Atari's arcade game Asteroids.

The game Asteroids illustrates the basic problem of collision detection and response in one of the simplest forms possible. Before, however, discussing object-to-object collision response as encountered in Asteroids, let's look at the game Breakout.

Breakout features a ball that can either bounce from the boundaries of the game window or movable paddle while also destroying bricks upon collision. Bouncing the ball off the screen boundaries requires very basic collision detection mainly because we already know where the boundaries of the screen are while at the same time only considering collisions with two horizontal and two vertical edges. Also, an object such as the ball in Breakout will always reflect at an angle equal and opposite to its initial incoming angle (illustrated in Figure E.10).

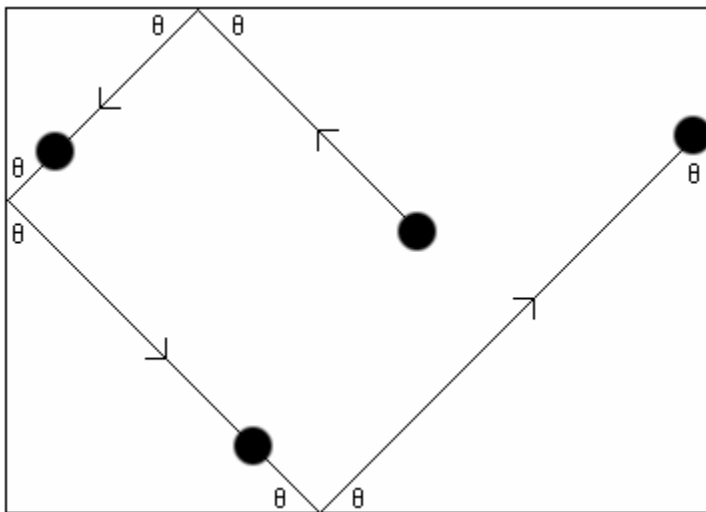


Figure E.10 A ball always reflects at an angle equal and opposite to its initial incoming angle.

Now, considering the shown image; it can be deduced that when the ball hits either vertical edge, then its direction can be changed by reversing the x-component of its velocity. Similarly, reversing the y-component of the ball's velocity upon collision with one of the horizontal edges will result in a perfect direction change:

```
/* initialise the object's initial position */
float objectXPos = 5;
float objectYPos = 2;
float objectZPos = 0;

/* set the object's initial velocity */
float objectXVelocity = 15;
float objectYVelocity = 20;
float objectZVelocity = 0;

/* update the object's velocity due to a vertical collision */
if(objectXPos > LEFT_EDGE || objectXPos < RIGHT_EDGE)
{
    /* update the object's velocity */
    objectXVelocity = -objectXVelocity;

    /* calculate the object's new x-position */
    objectXPos = objectXPos + objectXVelocity;
}

/*update the object's velocity due to a horizontal collision*/
if(objectYPos > BOTTOM_EDGE || objectYPos < TOP_EDGE)
{
    /* update the object's velocity */
    objectYVelocity = -objectYVelocity;

    /* calculate the object's new y-position */
    objectYPos = objectYPos + objectYVelocity;
}
```

This technique can now be extended to simulate one object bouncing off another. The simplest approach would be to test for horizontal and vertical collisions with the sides of a bounding volume. For example, consider the screenshot of the Asteroids clone in Figure E.11 where the bounding volume of each object is shown (these volumes are specified using the contained object's minimum and maximum x- and y-values).

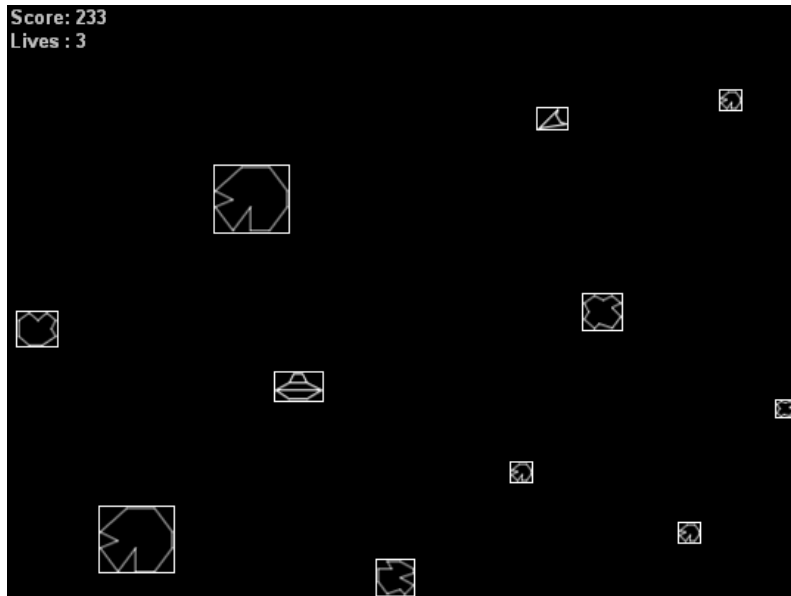


Figure E.11 Using bounding boxes to simulate inter-object collisions.

We can implement this approach in much the same way as with our horizontal and vertical screen boundary collision example – for instance, when we have a ball bouncing off objects as shown in Figure E.12, then we can change its direction by reversing the x-component of its velocity when it hits a vertical edge of another object. Similarly, reversing the y-component of the ball’s velocity upon collision with one of the horizontal edges will result in a perfect direction change.

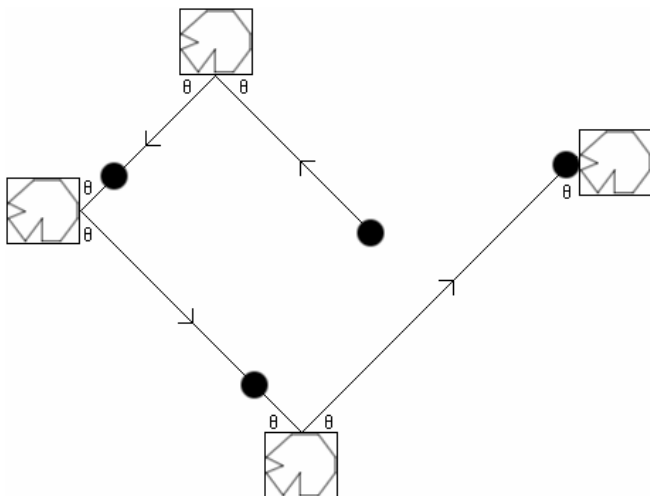


Figure E.12 Inter-object collisions – the same rules hold true as with screen boundary collisions.

The above given object collision approach works extremely well for horizontal and vertical surfaces, but in nearly all action-oriented games written today we’ll need to calculate vector reflections for arbitrarily rotated surfaces. For example, consider the

object shown in Figure E.13. This object has several flat planes, with each of these positioned at an arbitrary angle.

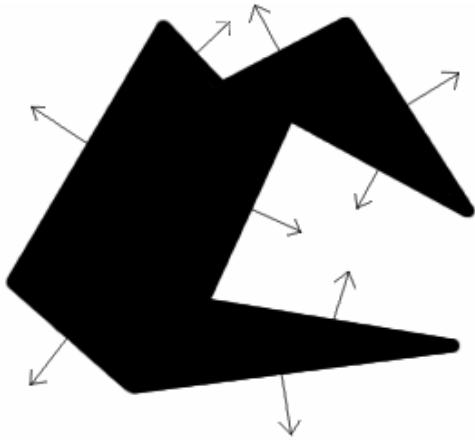


Figure E.13 An object with numerous arbitrarily positioned faces (the normal of each shown).

The core of collision detection, when dealing with arbitrarily positioned faces, is vector calculations; specifically the calculation of a reflection vector when we have an initial vector direction and a normal to the plane (Blinn, 1977). We've already looked at vector and normal calculations in Appendix C and will now look at an example to illustrate vector-based object reflections for arbitrarily rotated surfaces.

Figure E.14 illustrates our vector reflection problem; showing an incoming vector **I**, the surface normal **N** and the unknown reflection vector **R**.

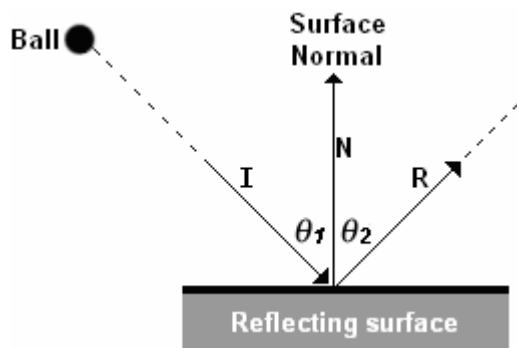


Figure E.14 Vector reflection for an arbitrarily rotated surface.

We use vector addition to create a third, composite vector. This process involves summing the related scalar components of two successive vectors (using the head-to-tail rule). In Figure E.14 we have three vectors, namely, **I**, **N** and **R**; using these vectors we define a third and fourth vector, **P** and **Q** (the resultant of **I** and **N** and **R** and **N** respectively) by summing the scalars of vector **I** and **N** and **R** and **N** in the following manner (graphically illustrated in Figure E.15):

$$\begin{aligned} \mathbf{P} &= \mathbf{I} + \mathbf{N} \\ &= (I_x + N_x, I_y + N_y, I_z + N_z) \\ &= (P_x, P_y, P_z). \end{aligned}$$

$$\begin{aligned} \mathbf{Q} &= \mathbf{R} + \mathbf{N} \\ &= (R_x + N_x, R_y + N_y, R_z + N_z) \\ &= (Q_x, Q_y, Q_z). \end{aligned}$$

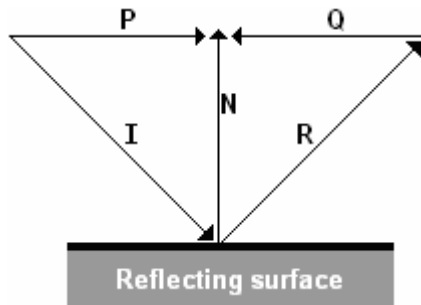


Figure E.15 The head-to-tail rule, creating a third composite vector.

Using the above given information, we can now algebraically calculate the reflection vector by stating that $\mathbf{P} = \mathbf{Q}$ and substituting the first equation into the second:

$$\begin{aligned} \mathbf{I} + \mathbf{N} &= \mathbf{R} + \mathbf{N} \\ \mathbf{R} &= \mathbf{N} + (\mathbf{I} + \mathbf{N}) \\ &= 2\mathbf{N} + \mathbf{I} \end{aligned}$$

Returning to our example, if the object has an incoming speed with an x-component of -16 and a y-component of 8 then we can calculate the vector of reflection (thus the exiting speed of the object) in the following manner (the normal in this case equals $y = 1$):

$$\begin{aligned} \mathbf{R} &= 2\mathbf{N} + \mathbf{I} \\ &= 2(-\mathbf{I} \cdot |\mathbf{N}|) * |\mathbf{N}| + \mathbf{I} \\ &= 2[(I_x, I_y) \cdot |(N_x, N_y)|] * |(N_x, N_y)| + (I_x, I_y) \\ &= 2[(-16, 8) \cdot |(0, 1)|] * |(0, 1)| + (-16, 8) \\ &= 2[(16, -8) \cdot |(0, 1)|] * |(0, 1)| + (-16, 8) \\ &= 2(16*0 - 8*1) * |(0, 1)| + (-16, 8) \\ &= 2(-8) * |(0, 1)| + (-16, 8) \\ &= -16*(0, -1) + (-16, 8) \\ &= (0, 16) + (-16, 8) \\ &= (0 - 16, 16 + 8) \\ &= (-16, 24). \end{aligned}$$

Collision detection and response in modern games often require considerable resources to implement. A number of collision detection algorithms (such as the detection of collisions using hierarchy trees) have consequently been developed to simulate collisions at various degrees of accuracy. The study of these algorithms is, however, beyond the scope of this thesis and our physics engine implementation.

The DXUT Framework

The Direct3D Utility Framework, or DXUT, is a high-level framework built on top of Direct3D. This framework provides a series of functions, call-backs, structures, constants and enumerations to reduce the complexity of low-level Direct3D routines. It encapsulates the Win32 and Direct3D APIs for ease of use, making it easier to create Direct3D applications. To summarise, DXUT allows simplified window creation, enables rapid Direct3D device setup and initialisation as well as the easy handling of Windows messages.

The DXUT framework provides a vast array of functionality, from basic window creation, Direct3D device initialisation and the control of these components to more advanced elements such as 3-D mesh control, camera control and the creation of graphical user interfaces. We will now look at the most important functional components provided by this framework.

The process of window creation and control using the DXUT framework is relatively simple when compared to using the Win32 API which entails creating a window, registering a window class, creating a window object and handling messages to and from the window. The DXUT framework simplifies this process in the sense that it is not necessary to register the window class (using the `WNDCLASSEX` structure) or to create the window using the `AdjustWindowRect`, `CreateWindow` and `ShowWindow` functions. The following series of DXUT function calls manages this entire window creation process:

```
/* initialise the DXUT framework */
DXUTInit(true, true, NULL);

/* configure mouse cursor settings for full-screen usage */
DXUTSetCursorSettings(true, true);

/* create the application window */
DXUTCreateWindow(L"DXUT Sample", NULL, NULL, NULL, NULL,
                NULL);

/* create the Direct3D device */
DXUTCreateDevice(true, 800, 600);

/* enter the main DXUT framework render loop */
DXUTMainLoop(NULL);
```


The `DXUTInit` function initialises DXUT by taking three parameters, namely a Boolean value used for the processing of command-line arguments (with the most common ones listed in Table F.1), another Boolean parameter controlling whether an error message box is to be displayed whenever an error occurs and a string value for the specification of additional command-line parameters.

Argument	Description
<code>-adapter:X</code>	Defines the specific hardware adapter to use.
<code>-automation</code>	Enables user interface navigation via the keyboard (enabled by default)
<code>-constantframetime</code>	Defines a specific time per frame lapse when the desired effect is to render some scene at a FPS value less than real-time.
<code>-forceapi:X</code>	Forces the application to use either the Direct3D 9 or Direct3D 10 API.
<code>-forcehal</code>	Forces the use of a HAL device type.
<code>-forcehwvp</code>	Forces the use of hardware vertex processing (not applicable for Direct3D 10 – only supported by the Direct3D 9 API).
<code>-forcepurehwvp</code>	Forces the use of pure hardware vertex processing (not applicable for Direct3D 10 – only supported by the Direct3D 9 API).
<code>-forceref</code>	Forces the use of a reference device type.
<code>-forceswvp</code>	Forces the use of software vertex processing (not applicable for Direct3D 10 – only supported by the Direct3D 9 API).
<code>-forcevsync:X</code>	Specifies whether vertical sync is to be used (<code>x</code> is set to '0' to disable vertical sync).
<code>-fullscreen</code>	Forces the application into full-screen mode on startup.
<code>-height:X</code>	Specifies the default window height.
<code>-noerrormsgboxes</code>	Disables DXUT's error message boxes.
<code>-nostats</code>	Disables the display of statistics such as the current number of frames per second.
<code>-output:X</code>	Forces the use of a specific adapter output (only supported by the Direct3D 10 API)
<code>-quitafterframe:X</code>	Sets an exit frame – i.e. forcing the application to terminate after the specified frame, <code>x</code> , has been rendered.
<code>-startx:#</code>	Sets the <code>x</code> -coordinate of the window's upper left corner when running in windowed mode.
<code>-starty:#</code>	Sets the <code>y</code> -coordinate of the window's upper left corner when running in windowed mode.

-width:x	Specifies the default window width.
-windowed	Forces the application into windowed mode on startup.

Table F.1 DXUTInit command-line parameters.

The next called function, **DXUTSetCursorSettings**, sets the visibility and clipping of the mouse cursor when used in full-screen mode. This function takes two parameters, the first a Boolean value specifying whether the mouse cursor will be visible for a window in full-screen mode (**true** if yes), and the second, also a Boolean value, defining whether the cursor will be limited from leaving the screen boundaries for a full-screen window (**true** if yes).

DXUTCreateWindow creates the application window through the initialisation of six parameters, namely, a string value defining the window’s caption, a **HINSTANCE** handle to the application instance (‘NULL’ by default), a **HICON** handle to the window’s icon (‘NULL’ by default), a **HMENU** handle to the window’s menu resource (‘NULL’ for no menu) and the upper left x- and y- window coordinates.

We create the actual Direct3D 10 device by calling the **DXUTCreateDevice** function. Its first parameter takes a Boolean value specifying whether the application will launch in windowed (**true**) or full-screen mode (**false**). **DXUTCreateDevice**’s final two parameters set the initial width and height of the back buffer, respectively.

The **DXUTMainLoop** function enters the main DXUT framework render loop (the main message loop), updating and rendering each frame via callbacks to the application. It takes one parameter, namely a handle to an accelerator table – this parameter is set to ‘NULL’ when no accelerator table is defined. *Accelerator tables* are created as resources and used for the translation of keyboard messages received from the message queue. One example of a common accelerator is the “Ctrl+S” key combination used as shortcut for the “File Save” menu item.

All these functions return the value “s_ok” if successful. In the event of a failure they return one of the error codes listed in Table F.2. Calling the **DXUTGetExitCode** function returns an exit code with ‘0’ indicating successful execution.

Error code	Description
DXUTERR_CREATINGDEVICE	Unable to create a Direct3D device.
DXUTERR_CREATINGDEVICEOBJECTS	A problem has been encountered while creating the Direct3D device objects.
DXUTERR_DEVICEREMOVED	The initialised Direct3D device is no longer accessible.
DXUTERR_MEDIANOTFOUND	The requisite media could not be loaded.

DXUTERR_NOCOMPATIBLEDEVICES	Unable to find any Direc3D capable devices.
DXUTERR_NODIRECT3D	Direct3D could not be initialised.
DXUTERR_NONZEROREFCOUNT	The Direct3D device was not properly released by a previous application.
DXUTERR_RESETTINGDEVICE	Unable to reset the Direct3D device.
DXUTERR_RESETTINGDEVICEOBJECTS	An issue was encountered while resetting the Direct3D device objects.

Table F.2 Error codes returned by DXUT functions.

Using the Win32 API, after registering the window class and creating the window, we enter the main message loop by declaring an empty `MSG` structure, `msg`, and passing it as parameter to the `WndProc` function. Using DXUT we no longer need to define a `MSG` structure or `WndProc` function for the handling of messages sent to and from the window. We will now rather create a series of callback functions, passing each one as a parameter to the appropriate `DXUTSetCallback*` DXUT function. For example, the following callback function handles all keyboard events:

```
void CALLBACK OnKeyPress(UINT nChar, bool bKeyDown,
                        bool bAltDown, void* pUserContext)
{
    /* test whether some key is being pressed */
    if(bKeyDown)
    {
        switch(nChar)
        {
            case VK_TAB: //if 'Tab' is pressed do something
                break;
        }
    }
}
```

This keyboard event callback function, `OnKeyPress`, is then passed as parameter to the `DXUTSetCallbackKeyboard` function:

```
DXUTSetCallbackKeyboard(OnKeyPress, NULL);
```

This function, initialising the previously defined callback function, takes two parameters, the first being a pointer to a `LPDXUTCALLBACKKEYBOARD` keyboard event callback function, and the second a pointer to some user-specific variable passed to the callback function – by default set to 'NULL'.

The `LPDXUTCALLBACKKEYBOARD` DXUT keyboard event callback function is called every time a keyboard event occurs. It is declared as follows in the `DXUT.h` header file:

```
VOID LPDXUTCALLBACKKEYBOARD(
    UINT nChar,
    bool bKeyDown,
    bool bAltDown,
    void* pUserContext
);
```

Its first parameter holds a virtual-key code describing the pressed key (the most commonly used virtual-key codes are given in Table F.3). The second parameter, `bKeyDown`, holds the Boolean value 'true' if a key is currently being pressed with the `bAltDown` parameter set to 'true' if the 'Alt' key is also being pressed. The last parameter, `pUserContext`, takes a pointer to a user-specific variable passed to the callback function – by default set to 'NULL'.

Constant	Description
<code>VK_LBUTTON</code>	Left mouse button.
<code>VK_RBUTTON</code>	Right mouse button.
<code>VK_BACK</code>	Backspace key.
<code>VK_TAB</code>	Tab key.
<code>VK_RETURN</code>	Enter key.
<code>VK_ESCAPE</code>	Escape key.
<code>VK_UP</code> , <code>VK_DOWN</code> , <code>VK_LEFT</code> , <code>VK_RIGHT</code>	Up, down, left and right keys respectively.
<code>VK_NUMPAD0</code> to <code>VK_NUMPAD9</code>	Numeric keypad keys '0' to '9'.
<code>VK_F1</code> to <code>VK_F24</code>	F1 to F24 keys.

Table F.3 Virtual-Key codes.

DXUT provides a number of these so-called application-defined callback functions. The above defined `onKeyPress` function is, for example, a `LPDXUTCALLBACKKEYBOARD` keyboard event callback. These DXUT event callback functions simplify the message handling process. In addition to a keyboard event callback we also have to define a device acceptable callback function (set using the `DXUTSetCallbackD3D10DeviceAcceptable` DXUT initialisation function), a device created callback function (set via `DXUTSetCallbackD3D10DeviceCreated`), a swap chain resized callback function (set using `DXUTSetCallbackD3D10SwapChainResized`), a swap chain release callback function (set via `DXUTSetCallbackD3D10SwapChainReleasing`), a device destroyed callback function (set via `DXUTSetCallbackD3D10DeviceDestroyed`) and a frame render callback function (set through the `DXUTSetCallbackD3D10FrameRender` DXUT initialisation function). In addition to

these callback functions we also need to create a window message callback function dealing with Windows messages (set using `DXUTSetCallbackMsgProc`), a callback function dealing with frame updates (set by `DXUTSetCallbackFrameMove`) and a callback function that allows for the change of device settings before the creation of the device (set through the `DXUTSetCallbackDeviceChanging` DXUT initialisation function).

`DXUTSetCallbackD3D10DeviceAcceptable` initialises the application specific callback function responsible for building an enumerated list of Direct3D 10 capable devices. It takes two parameters, namely, a pointer to a `LPDXUTCALLBACKISD3D10DEVICEACCEPTABLE` callback function and a pointer to a user-defined variable passed to the callback function – ‘NULL’ by default.

The `LPDXUTCALLBACKISD3D10DEVICEACCEPTABLE` callback function returns true for each acceptable Direct3D device. All acceptable Direct3D 10 devices are enumerated into a list by the `DXUTSetCallbackD3D10DeviceAcceptable` function. DXUT then selects the best rendering device from this list. This callback is declared as follows in the DXUT.h header file:

```
bool LPDXUTCALLBACKISD3D10DEVICEACCEPTABLE(
    UINT Adapter,
    UINT Output,
    D3D10_DRIVER_TYPE DeviceType,
    DXGI_FORMAT BackBufferFormat,
    bool bWindowed,
    void* pUserContext
);
```

Its first parameter, `Adapter`, holds a value indicating the position of the current Direct3D 10 device in a series of enumerated Direct3D 10 video adapters. The second parameter, `Output`, holds an index value of the current enumerated video adapter’s output (such as a monitor). The `DeviceType` parameter holds the current Direct3D 10 capable video adaptor’s driver type (commonly set to `D3D10_DRIVER_TYPE_HARDWARE` for a hardware device and `D3D10_DRIVER_TYPE_REFERENCE` for a reference device). The `BackBufferFormat` parameter indicates the back buffer format of the Direct3D 10 device (such as a four-component, 64-bit floating-point format). The next parameter takes a Boolean value that is set to ‘true’ for windowed application and ‘false’ for those running in full-screen mode. The final parameter, `pUserContext`, is a pointer to a user-specific variable passed to the callback function – ‘NULL’ by default unless context information for the callback function is needed.

We create a `LPDXUTCALLBACKISD3D10DEVICEACCEPTABLE` callback function which is passed as first parameter to the `DXUTSetCallbackD3D10DeviceAcceptable` DXUT function as follows:

```
/* return 'true' for all acceptable D3D10 devices passed to it */
bool CALLBACK OnDeviceAcceptable(UINT Adapter, UINT Output,
                                  D3D10_DRIVER_TYPE DeviceType,
                                  DXGI_FORMAT BufferFormat,
                                  bool bWindowed, void* pUserContext )
{
    return true;
}

DXUTSetCallbackD3D10DeviceAcceptable(OnDeviceAcceptable,NULL);
```

The `DXUTSetCallbackD3D10DeviceCreated` function sets the created `ID3D10Device` device. This device interface is used for the rendering of primitives as well as the creation of shaders and resources. The callback is used for the allocation of resources and the initialisation of buffers. The `DXUTSetCallbackD3D10DeviceCreated` function takes two parameters, namely, a pointer to a `LPDXUTCALLBACKD3D10DEVICECREATED` callback function and a pointer to a user-define variable passed to the callback function – ‘NULL’ by default. This function is declared as follows:

```
VOID DXUTSetCallbackD3D10DeviceCreated(
    LPDXUTCALLBACKD3D10DEVICECREATED pCallback,
    void* pUserContext
);
```

The associated `LPDXUTCALLBACKD3D10DEVICECREATED` callback function is declared as follows:

```
HRESULT LPDXUTCALLBACKD3D10DEVICECREATED(
    ID3D10Device * pd3dDevice,
    CONST DXGI_SURFACE_DESC * pBackBufferSurfaceDesc,
    void* pUserContext
);
```

This resource callback function forwards a pointer to the newly created `ID3D10Device` interface – the Direct3D 10 device. This pointer, sent to the `DXUTSetCallbackD3D10DeviceCreated` function, is defined as the first parameter. The second parameter is a `DXGI_SURFACE_DESC` structure with four members describing the width, height, format and multisampling parameters of the surface

resource respectively. The third parameter, `pUserContext`, is a pointer to a user-specific variable passed to the callback function – ‘NULL’ by default unless context information for the callback function is needed.

A `LPDXUTCALLBACKD3D10DEVICECREATED` callback function can be defined in the following manner:

```
HRESULT CALLBACK OnCreateDevice(ID3D10Device* pd3dDevice,
                               const DXGI_SURFACE_DESC *pBufferSurfaceDesc,
                               void* pUserContext)
{
    /* - set up, create and set the input layout
       - create and set the vertex buffer
       - create and set the index buffer
       - specify the primitive topology
       - load all texture resources
       - initialise the world and view matrices */
}
```

This function is now set using the `DXUTSetCallbackD3D10DeviceCreated` DXUT initialisation function:

```
DXUTSetCallbackD3D10DeviceCreated(OnCreateDevice, NULL);
```

We also have to deal with the callbacks sent to the application whenever the Direct3D 10 swap chain is resized (see section 4.5.2), this is done using the `DXUTSetCallbackD3D10SwapChainResized` function. This function has two parameters, the first a pointer to a `LPDXUTCALLBACKD3D10SWAPCHAINRESIZED` callback function with the second a pointer to a user-specific variable passed to the callback function – ‘NULL’ by default.

The `LPDXUTCALLBACKD3D10SWAPCHAINRESIZED` callback function commonly used to set resources dependent on the back buffer – such as perspective projection matrices based on the field-of-view is declared as follows:

```
HRESULT LPDXUTCALLBACKD3D10SWAPCHAINRESIZED(
    ID3D10Device * pd3dDevice,
    IDXGISwapChain * pSwapChain,
    CONST D3DSURFACE_DESC * pBackBufferSurfaceDesc,
    void* pUserContext
);
```


Its first parameter, `pd3dDevice`, is a pointer to the newly created Direct3D 10 device (`ID3D10Device`). The second parameter is a pointer to an `IDXGISwapChain` interface (see section 4.5.2) with the third holding a pointer to a structure describing the back buffer surface's format. The last parameter, `pUserContext`, is a pointer to a user-specific variable passed to the callback function.

This `LPDXUTCALLBACKD3D10SWAPCHAINRESIZED` swap chain resized callback function, passed to `DXUTSetCallbackD3D10SwapChainResized`, can be defined in the following manner:

```
HRESULT CALLBACK OnSwapChainResize(ID3D10Device* pd3dDevice,
                                   IDXGISwapChain *pSwapChain,
                                   const DXGI_SURFACE_DESC* pBufferSurfaceDesc,
                                   void* pUserContext)
{
    /* - reset the aspect ratio using the back buffer's new width and height
       - set the perspective projection matrix using the
          new aspect ratio */
}
```

We set this callback function using `DXUTSetCallbackD3D10SwapChainResized`:

```
DXUTSetCallbackD3D10SwapChainResized(OnSwapChainResize);
```

All the Direct3D 10 device resources created in the `LPDXUTCALLBACKD3D10SWAPCHAINRESIZED` callback function must also be released. This is done using a `LPDXUTCALLBACKD3D10SWAPCHAINRELEASING` callback which is set using the `DXUTSetCallbackD3D10SwapChainReleasing` swap chain releasing function. This DXUT function takes two parameters, a pointer to a `LPDXUTCALLBACKD3D10SWAPCHAINRELEASING` callback function and a pointer to a user-specific variable passed to the callback function – 'NULL' by default:

```
HRESULT DXUTSetCallbackD3D10SwapChainReleasing(
    LPDXUTCALLBACKD3D10SWAPCHAINRELEASING pCallback,
    void* pUserContext
);
```

The `LPDXUTCALLBACKD3D10SWAPCHAINRELEASING` callback function has only one parameter, a pointer to a user-specific variable passed to the callback function when context information for the callback function is needed:

```
VOID LPDXUTCALLBACKD3D10SWAPCHAINRELEASING(
    void* pUserContext
```

```
);
```

This **LPDXUTCALLBACKD3D10SWAPCHAINRELEASING** swap chain releasing callback function, called whenever the swap chain created in **OnSwapChainResize** is being released can be defined as follows:

```
void CALLBACK OnSwapChainReleasing(void* pUserContext )
{
    /* release all the Direct3D 10 resources created in
       OnSwapChainResize */
}
```

We can now set the **OnSwapChainReleasing** callback function via the **DXUTSetCallbackD3D10SwapChainReleasing** DXUT function:

```
DXUTSetCallbackD3D10SwapChainReleasing(OnSwapChainReleasing);
```

We also require a callback function to release the Direct3D 10 resources created in the **OnCreateDevice** callback function. This resource deletion callback, **LPDXUTCALLBACKD3D10DEVICEDESTROYED**, is executed by the DXUT framework immediately after the Direct3D 10 device has been destroyed. The **DXUTSetCallbackD3D10DeviceDestroyed** function, with its first parameter taking a pointer to a **LPDXUTCALLBACKD3D10DEVICEDESTROYED** function, sets the device destroyed callback. Its second parameter is a pointer to a user-specific variable passed to the callback function whenever context information is needed:

```
VOID DXUTSetCallbackD3D10DeviceDestroyed(
    LPDXUTCALLBACKD3D10DEVICEDESTROYED pCallback,
    void* pUserContext
);
```

The **LPDXUTCALLBACKD3D10DEVICEDESTROYED** callback function specifies only one parameter, namely a pointer to a user-specific variable for the gathering of context information, **pUserContext**:

```
VOID LPDXUTCALLBACKD3D10DEVICEDESTROYED(
    void* pUserContext
);
```

A **LPDXUTCALLBACKD3D10DEVICEDESTROYED** resource deletion callback function can be defined as follows:

```
void CALLBACK OnDeviceDestroy(void* pUserContext)
```

```
{
    /* release all the Direct3D 10 resources created in the
       OnCreateDevice callback function */
}
```

This callback function is then subsequently set using the `DXUTSetCallbackD3D10DeviceDestroyed` DXUT function:

```
DXUTSetCallbackD3D10DeviceDestroyed(OnDeviceDestroy);
```

Another significant DXUT callback function is one that deals with frame rendering. This `LPDXUTCALLBACKD3D10FRAMERENDER` callback function renders a scene using the created Direct3D 10 device by clearing the back buffer, depth-stencil buffer, updating all variable changes per frame and rendering the geometric objects constituting the scene. This function has four parameters and is declared as follows in the DXUT.h header file:

```
VOID LPDXUTCALLBACKD3D10FRAMERENDER(
    ID3D10Device * pd3dDevice,
    DOUBLE fTime,
    FLOAT fElapsedTime,
    void* pUserContext
);
```

Its first parameter, `pd3dDevice`, is a pointer to an `ID3D10Device` interface – the rendering device. The second parameter, `fTime`, holds the time that has elapsed since initialisation of the application with the third parameter, `fElapsedTime`, holding the time that has passed since the last frame update. Both these time values are given in seconds. The final parameter holds a pointer to the user-specific variable that is passed to the callback function whenever context information is needed. Just as with all the other DXUT callback functions, we will also set this one to ‘NULL’.

Such a `LPDXUTCALLBACKD3D10FRAMERENDER` callback function can be declared as follows:

```
void CALLBACK OnRenderFrame(
    ID3D10Device* pd3dDevice,
    double fTime, float fElapsedTime,
    void* pUserContext)
{
    /* - clear the back buffer using ClearRenderTargetView
       - clear the depth-stencil buffers using
           ClearDepthStencilView
       - update all changed variables
       - render all geometric objects */
```

```
}
```

This `OnRenderFrame` callback function is set by the `DXUTSetCallbackD3D10FrameRender` function:

```
DXUTSetCallbackD3D10FrameRender(OnRenderFrame);
```

All that remains now is to handle all process messages originating from the DXUT message pump and to set the callback function responsible for doing the frame updates for the scene. We also require a facility that allows us to change the settings of a device before it is created.

Processing messages for the DXUT message pump requires the declaration of a `LPDXUTCALLBACKMSGPROC` callback function similar to the previously defined `WinProc` function. This function takes six parameters, the first being a handle to the window, the second an integer value identifying the message to process, the third and fourth parameters specifying additional message information, with the fifth a Boolean value that controls whether further message processing should be done ('true' preventing further message handling). The final parameter is a pointer to a user-specific variable passed to the callback function whenever context information is needed:

```
LRESULT LPDXUTCALLBACKMSGPROC(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam,
    bool * pbNoFurtherProcessing,
    void* pUserContext
);
```

We can declare a `LPDXUTCALLBACKMSGPROC` callback function as follows:

```
LRESULT CALLBACK MsgProcCallback( HWND hWnd, UINT uMsg,
                                   WPARAM wParam,
                                   LPARAM lParam,
                                   bool* pbNoFurtherProcessing,
                                   void* pUserContext)
{
    /* handle all messages sent to the application */
}
```

The `DXUTSetCallbackMsgProc` DXUT function sets this window message callback function with its first parameter a pointer to the `LPDXUTCALLBACKMSGPROC` function and

its second a pointer to a user-specific variable passed to the callback function whenever context information is needed:

```
DXUTSetCallbackMsgProc(MsgProcCallback);
```

Frame updates of the scene are done via the **LPDXUTCALLBACKFRAMEMOVE** callback function. This function takes three parameters, namely, the time that has elapsed since initialisation of the application, the time elapsed since the previous frame and a pointer to a user-specific variable passed to the callback function whenever context information is needed:

```
VOID LPDXUTCALLBACKFRAMEMOVE(
    DOUBLE fTime,
    FLOAT fElapsedTime,
    void* pUserContext
);
```

Such a **LPDXUTCALLBACKFRAMEMOVE** callback function handling updates to a scene can be declared as follows:

```
void CALLBACK OnMoveFrame( double fTime, float fElapsedTime,
                          void* pUserContext)
{
    /* update the scene */
}
```

This callback function is subsequently set using the **DXUTSetCallbackFrameMove** DXUT function:

```
DXUTSetCallbackFrameMove(OnMoveFrame, NULL);
```

One final callback function is needed for the modification of Direct3D device settings as required. This callback function, **LPDXUTCALLBACKMODIFYDEVICESETTINGS**, takes a pointer to a **DXUTDeviceSettings** structure storing the settings of our Direct3D 10 device, and a pointer to a user-specific variable passed to the callback function whenever context information is needed:

```
bool LPDXUTCALLBACKMODIFYDEVICESETTINGS(DXUTDeviceSettings * pDeviceSettings,
                                         void* pUserContext);
```

An example of a **LPDXUTCALLBACKMODIFYDEVICESETTINGS** callback function is given here:

```
bool CALLBACK ModDevSettings(DXUTDeviceSettings* pDeviceSettings, void* pUserContext)
{
    /* allow modification of device settings */
    return true;
}
```

This callback function is called just before the creation of the Direct3D device. It returns a 'true' indicating that DXUT can proceed to create the device, and a 'false' indicating otherwise. The `DXUTSetCallbackDeviceChanging` function sets this callback function, allowing the application program to modify the device settings as needed. This function takes two parameters, a pointer to a `LPDXUTCALLBACKMODIFYDEVICESETTINGS` callback function and a pointer to a user-specific variable passed to the callback function whenever context information is needed:

```
DXUTSetCallbackDeviceChanging(ModDevSettings, NULL);
```

The functions presented in this section illustrate the fundamentals of the DXUT framework. This framework is useful for experimental applications where the desire is to minimise the amount of time spent on setting up a Direct3D environment. Although the DXUT framework's effectiveness in the simplification of Direct3D API calls cannot be disputed, it must be used with utmost caution as it does impose some level of performance overhead.