UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Part I

Introduction and Implementation

Chapter 1

Introduction

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

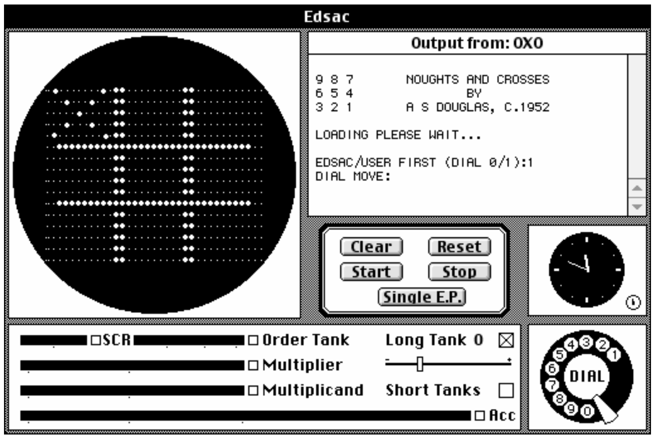
Outline:

- 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.



```
9 8 7       NOUGHTS AND CROSSES
6 5 4               BY
3 2 1       A S DOUGLAS, C.1952

LOADING PLEASE WAIT...

EDSAC/USER FIRST (DIAL 0/1):1
DIAL MOVE:
```

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.

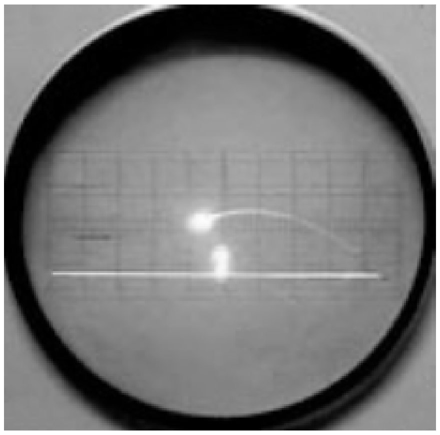


Figure 1.2 A photograph of the game Tennis for Two.

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.

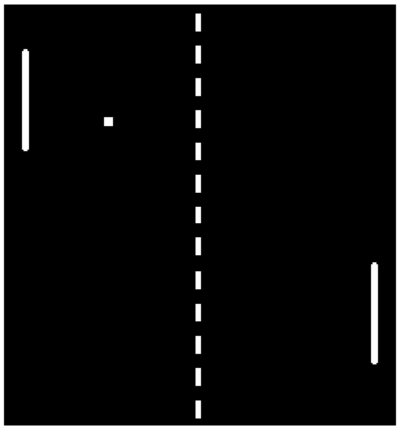


Figure 1.3 A PONG clone.

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 *Hovertank 3D* and *Catacomb 3D* during 1991. Other older PC games such as *Elite* also featured 3D environments but never achieved the level of technical complexity of Wolfenstein 3D nor its cultural and industry impact. Another breakthrough in the graphics of 3D games came with id Software's release of Doom in 1993. Doom, a screenshot of which is shown in Figure 1.8, really revolutionised the gaming industry (GameSpy, 2001) with its fast paced network play and immersive graphics and companies like Microsoft started spending millions of dollars on research and development to migrate gaming from MS-DOS to their Windows platform (Craddock, 2007). This research and development culminated in the DirectX Application Programming Interface (API).

Figure 1.8 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 stales 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 specularity. However, disabling these features resulted in a less than satisfactory gaming experience. Shadows and other special effects such as specular highlights and real-time reflections have become expected, and today's mid-range hardware is more than adequate in handling each of these effects separately. However, the performance impact remains an issue when real-time 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 it's 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 show volumes and distant objects via blop 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 it's 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 el al's algorithm based on the elimination of various shadow volume testing phases and our own algorithm based on shadow volumes, spatial subdivision and instruction set utilisation. 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 el 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
▪ Akenine-Möller T. and Assarsson U. (2002)	▪ Alard J. and Raffin B. (2005)	▪ August D., Huang J., Jablin T., Kim H., Mason T., Prabhu P., Raman A. and Zhang Y (2011)	▪ Belleman R., Bedorf J., Zwart S. (2008)
▪ Atherton P., Weiler K. and Greenberg D. (1978)	▪ Angel E. (2006)	▪ Epic Games (2012)	▪ Choppin B. (2004)
▪ Bergeron, P. (1985)	▪ AMD (2011)	▪ Harbour J.S. (2004)	▪ Crossno P. and Angel E. (1997)
▪ Blinn J. (1988)	▪ Bier E. and Sloan K. (1986)	▪ Hermann E., Raffin B., Faure F., Gautier T., Allard J. (2011)	▪ Flynt J. and Salem O. (2004)
▪ Bouknight W. and Kelly K. (1970)	▪ Blinn J. (1977)	▪ Fangerau J., Krömker S. (2010)	▪ Funge J. (1999)
▪ Brabec S. and Seidel H. (2002)	▪ Blinn J. and Newell M. (1976)	▪ Fernando R. (2004)	▪ Giarratano J., Riley G. (2005)
▪ Brotman L.S. and Badler N.I. (1984)	▪ Bouknight W. and Kelly K. (1970)	▪ Future Chips (2011)	▪ Hahn J. (1988)
▪ Carmack J. (2000)	▪ Boulanger K., Pattanaik S. and Bouatouch K. (2006)	▪ Huang J., Raman A., Zhang Y., Jablin T., Hung T., and August D. (2010)	▪ Halliday D., Resnick R. and Walker J. (2007)
▪ Chan E. and Durand F. (2004)	▪ Cabral B., Max N. and Springmeyer R. (1987)	▪ Id Software. (2011)	▪ Hecker C. (2000)
▪ Crow F. (1977)	▪ Cohen et al. (1998)	▪ Intel. (2011)	▪ Hermann E., Raffin B., Faure F., Gautier T., Allard J. (2011)
▪ Dimitrov R. (2007)	▪ Crytek 2 (2011)	▪ Jablin T., Prabhu P., Jablin J., Johnson N., Beard S., August D. (2011)	▪ Hubbard P. (1996)
▪ Drettakis G. and Fiume E. (1994)	▪ Drettakis G. and Fiume E. (1994)	▪ Jablin T., Jablin J., Prabhu P., Liu F, and August D. (2012)	▪ Ignizio J. (1991)
▪ Everitt C., Rege A. and Cebenoyan C. (2001)	▪ Gray K. (2003)	▪ Joselli M., Zamith M., Clua E., Montenegro A., Leal-Toledo R., Conci A., Pagliosa P., Valente L., Feijó B. (2009)	▪ Joselli M., Clua E., Montenegro A., Conci A., Pagliosa P. (2008)
▪ Everitt C. and Kilgard M. (2002)	▪ Goral C, Torrance D., Greenberg D. and Battaile B. (1984)	▪ Moore S. (2011)	▪ Lay D. (2005)
▪ Fernando R., Fernandez S., Bala K. and Greenberg D. (2001)	▪ Greene N. (1986)	▪ Nickolls J.. Kirk D. (2009)	▪ Mamdani E. H., Assilian S. (1975)
▪ Haines E. (2001)	▪ Heckbert P. (1986)		▪ Moore M. and Wilhelms J.
▪ Heidmann T. (1991)	▪ Hearn D. and Baker M. (2004)		
▪ Heidrich W., Brabec S. and Seidel H. (2000)	▪ Kalogirou, H. (2006)		
▪ Hourcade J.-C. and Nicolas A. (1985)	▪ Landis, McGaugh and Koch (2010)		
▪ Isard M., Shand M., and Heirich A. (2002)	▪ Langer, Bülthoff (2000)		
▪ Kersten D., Mamassian P. and Knill D. (1994)	▪ Levoy M. and Hanrahan P. (1996)		
▪ Kersten D., Mamassian P. and Knill D. (1997)	▪ Microsoft (2006 – 2011)		
	▪ Mikkelsen M. (2008)		
	▪ Nguyen H. (2007)		
	▪ NVIDIA (2009-2011)		

▪ 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)	▪ 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)	▪ 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)	(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 as 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.

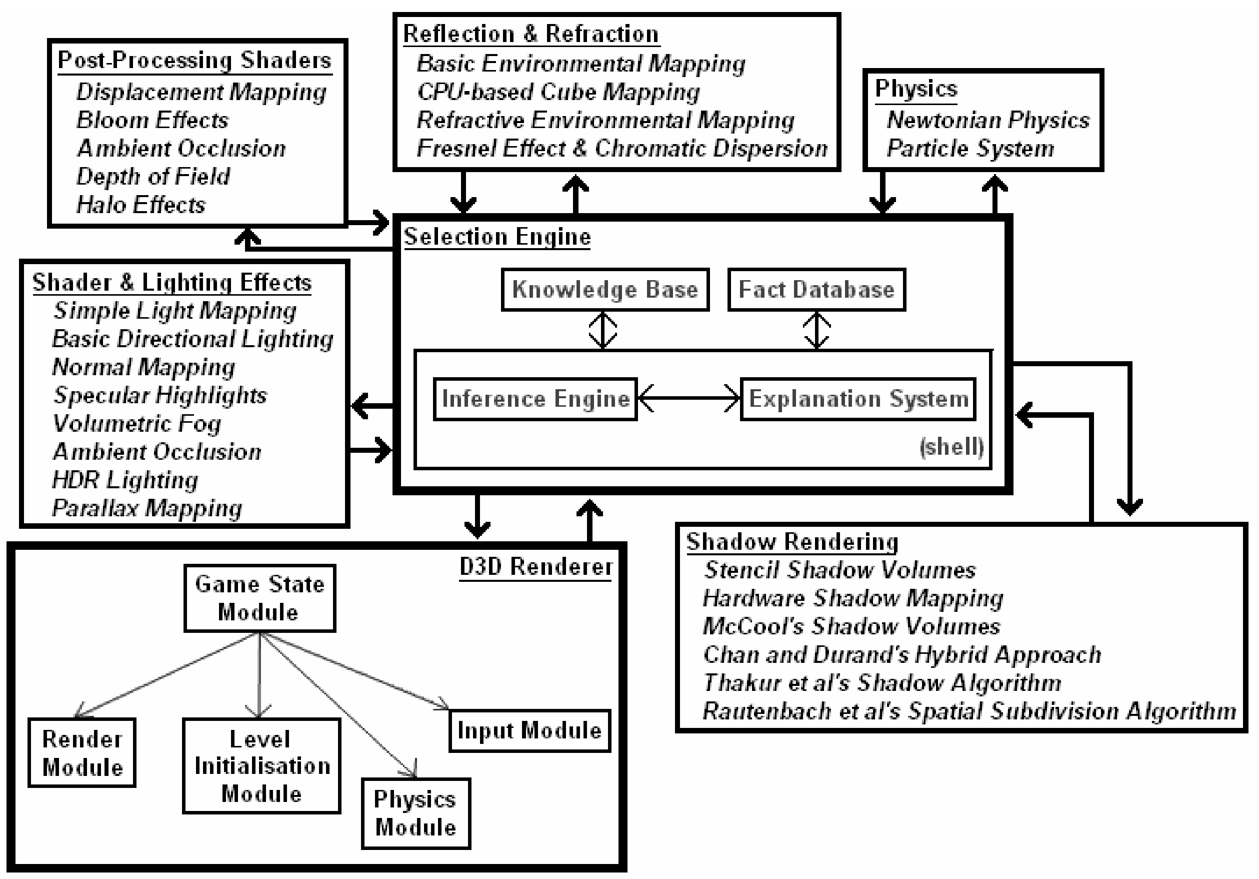


Figure 1.9 An abstract model illustrating the generality of the proposed system.

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).

Chapter 2

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.

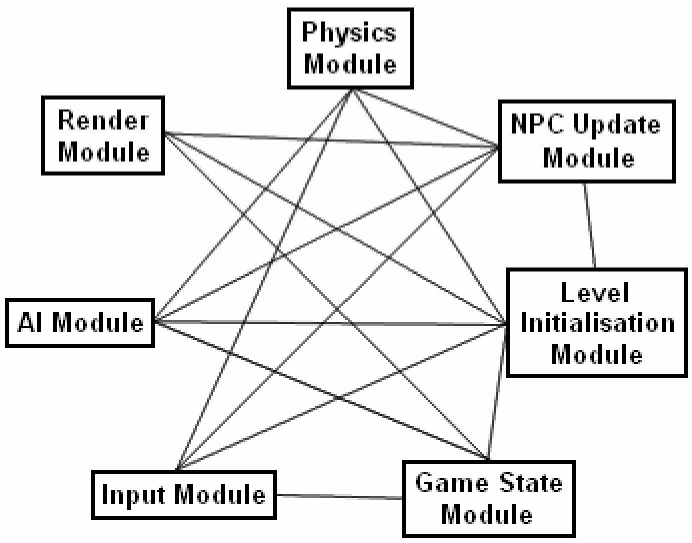


Figure 2.1 A modular architecture.

A *directed acyclic graph architecture* is a modular architecture where the inter-module dependencies are strictly regulated. A direct 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 that others. This hierarchical structure, shown in Figure 2.2, ensures relative loose coupling.

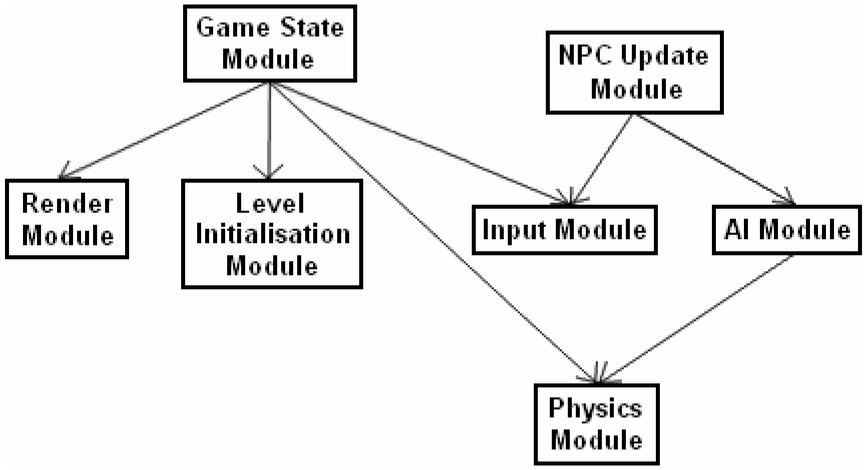


Figure 2.2 A directed acyclic graph architecture.

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 lever 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();
}
```

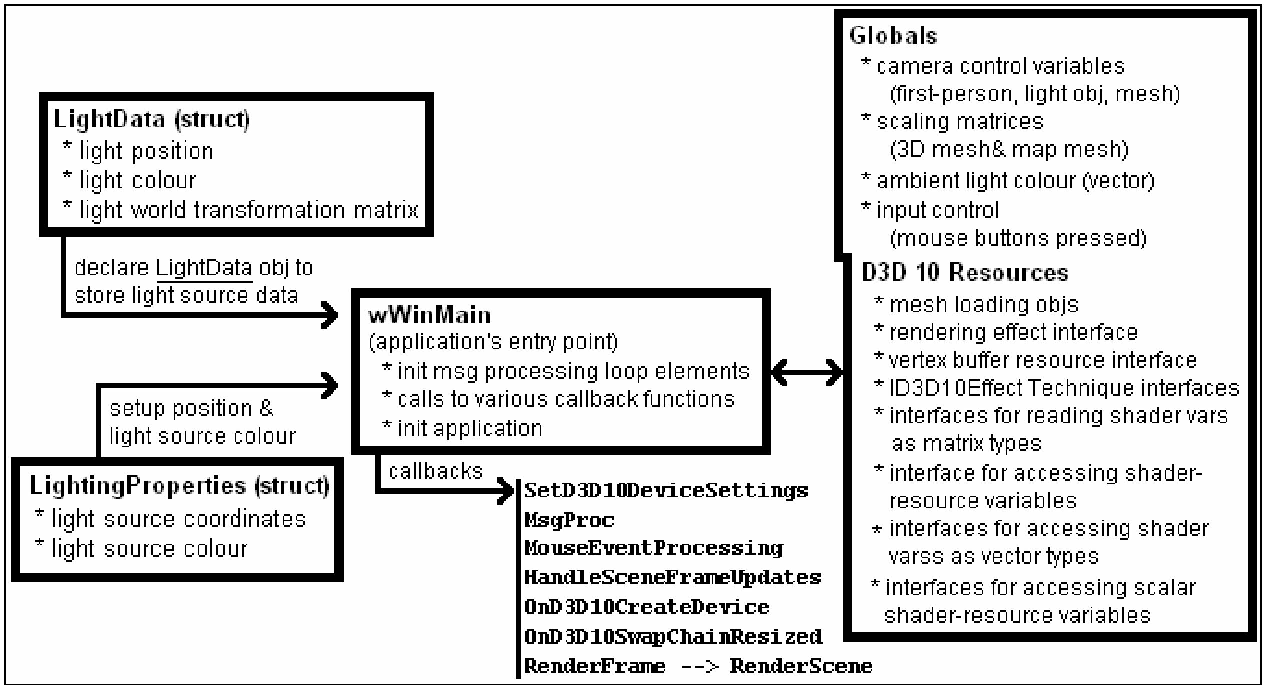
We can often improve performance by decoupling the game loop's rendering step from all the other update tasks. This will result in the rendering phase updating at a much higher rate than the other steps, however, all this will accomplish is several duplicate frames for each slower update. This situation 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();
}
```

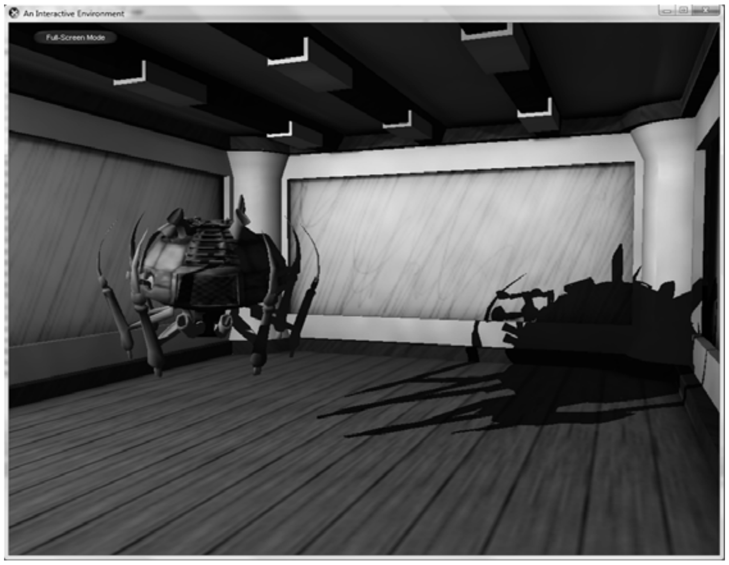
There are also numerous other miscellaneous tasks that can be performed during the game loop. Of course tasks such as network processing are not needed for single player game modes and should be removed from the game loop to improve performance. The easiest way of doing this is to add a check at the 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


LightData (struct)
* light position
* light colour
* light world transformation matrix
declare LightData obj to store light source data
setup position & light source colour
LightingProperties (struct)
* light source coordinates
* light source colour
wWinMain
(application's entry point)
* init msg processing loop elements
* calls to various callback functions
* init application
callbacks
SetD3D10DeviceSettings
MsgProc
MouseEventProcessing
HandleSceneFrameUpdates
OnD3D10CreateDevice
OnD3D10SwapChainResized
RenderFrame --> RenderScene
Globals
* camera control variables (first-person, light obj, mesh)
* scaling matrices (3D mesh& map mesh)
* ambient light colour (vector)
* input control (mouse buttons pressed)
D3D 10 Resources
* mesh loading objs
* rendering effect interface
* vertex buffer resource interface
* ID3D10Effect Technique interfaces
* interfaces for reading shader vars as matrix types
* interface for accessing shader-resource variables
* interfaces for accessing shader varss as vector types
* interfaces for accessing scalar shader-resource variables

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_FPSModelViewCamera.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).

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.

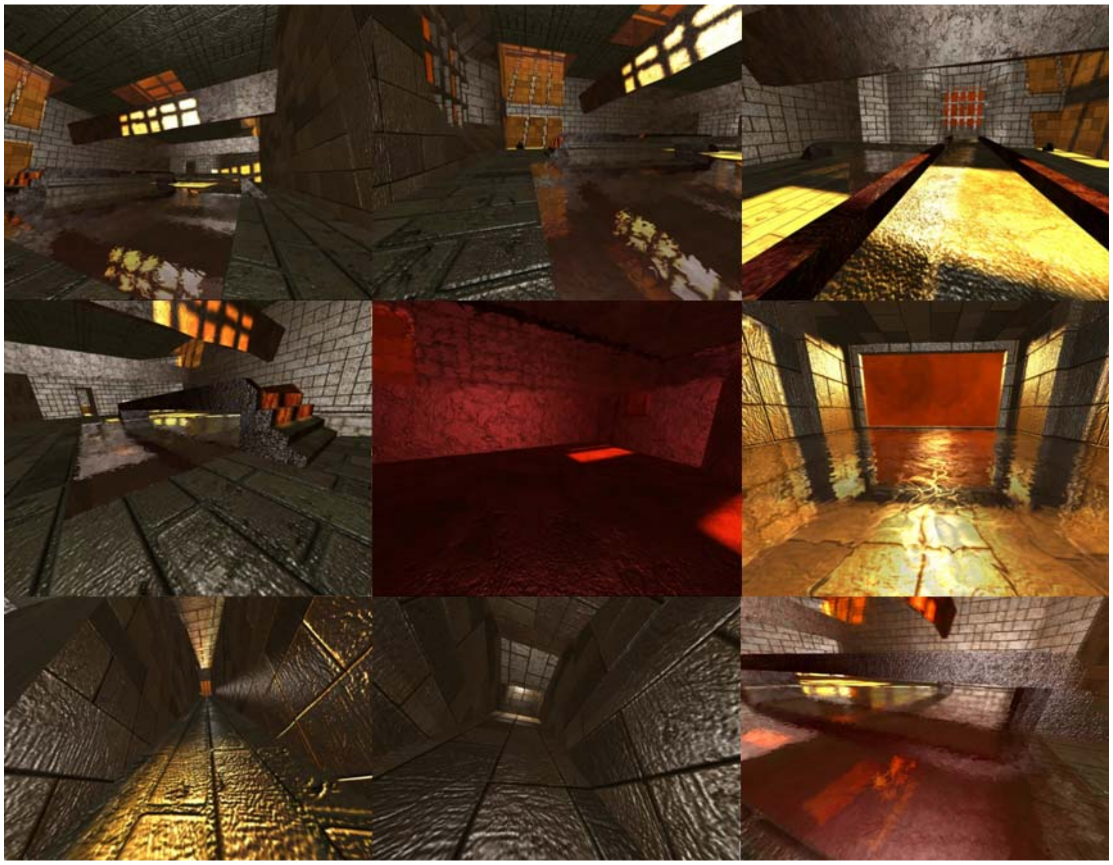


Figure 3.1 Various screenshots of the extended interactive testing environment.

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.

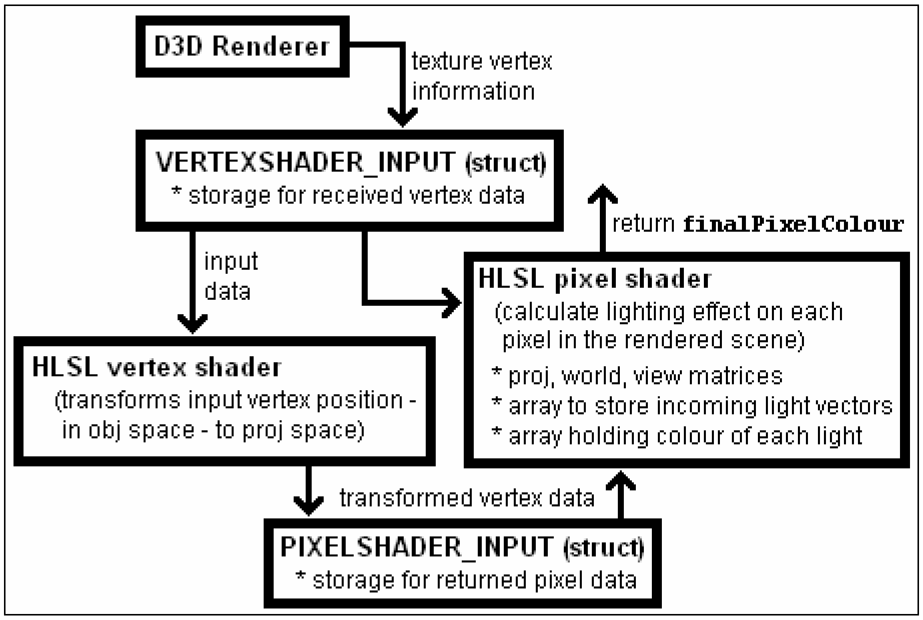
These three types of shaders are unified by the Direc3D 10 architecture – known as Shader Model 4.0. *Unified shaders* provide the application programmer with a uniform instruction set that is independent of whether a pixel, geometry or vertex shader is being implemented. This unified architecture is 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 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`**, 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


D3D Renderer
texture vertex information
VERTEXSHADER_INPUT (struct)
* storage for received vertex data
return finalPixelColour
input data
HLSL pixel shader
(calculate lighting effect on each pixel in the rendered scene)
* proj, world, view matrices
* array to store incoming light vectors
* array holding colour of each light
HLSL vertex shader
(transforms input vertex position - in obj space - to proj space)
transformed vertex data
PIXELSHADER_INPUT (struct)
* storage for returned pixel data

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.

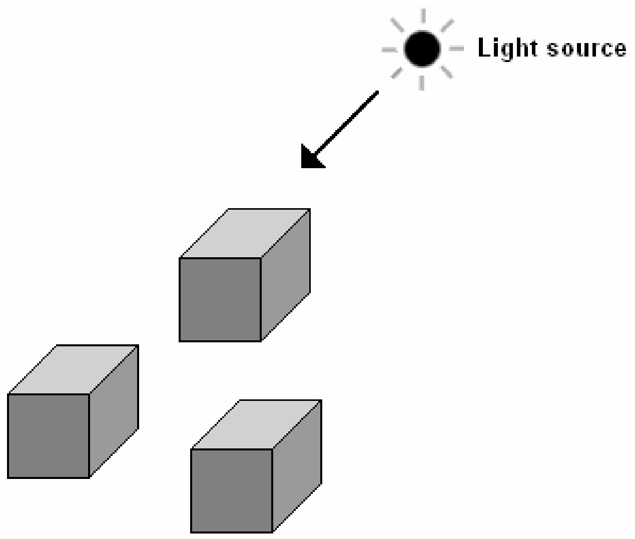


Figure 3.2 The local illumination model.

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.

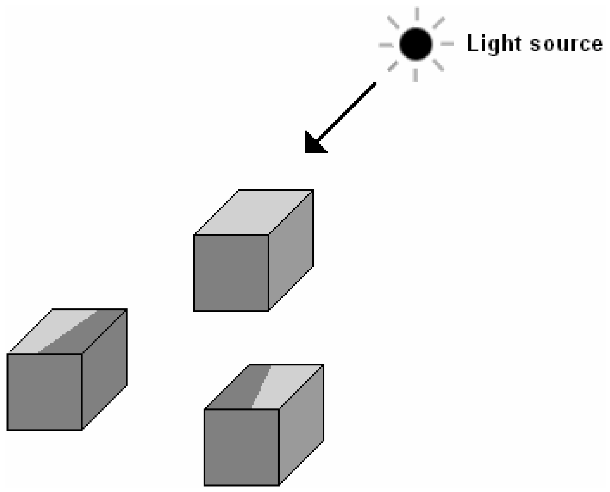


Figure 3.3 The global illumination model.

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).

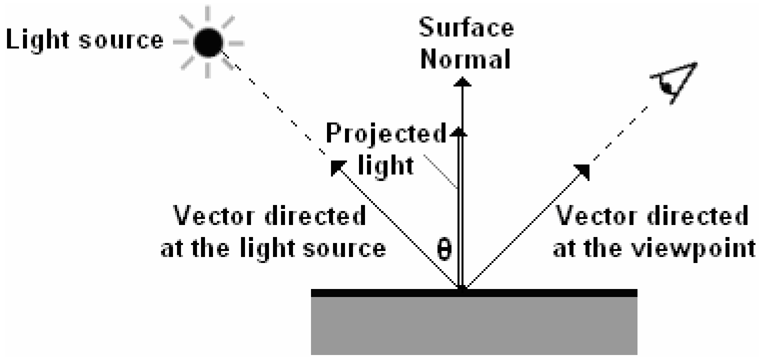

Light source
Surface Normal
Projected light
Vector directed at the light source
θ
Vector directed at the viewpoint

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).

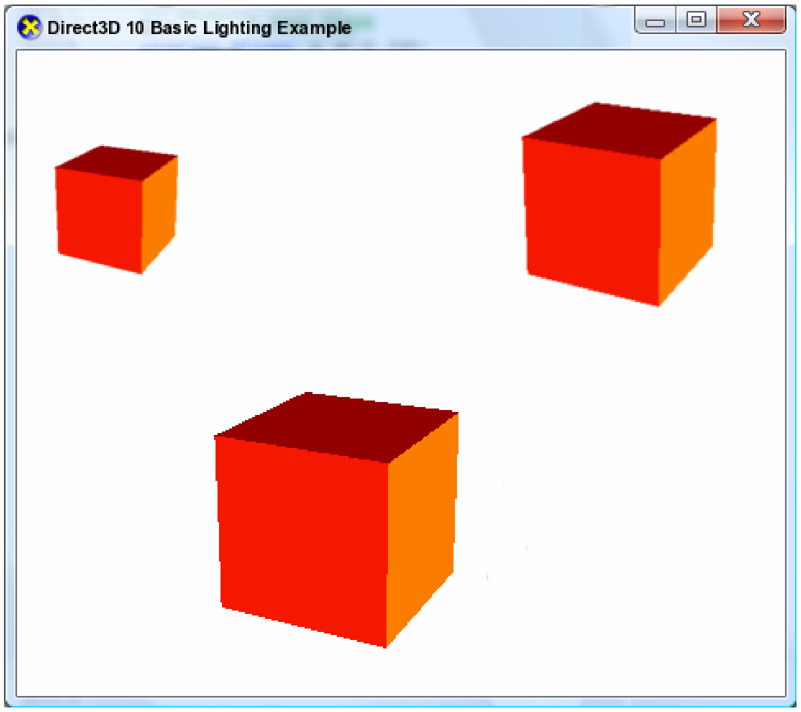


Figure 3.5 The rendered scene; comprised of three cubes, two parallel light sources and using Lambertian light.

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 then 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".

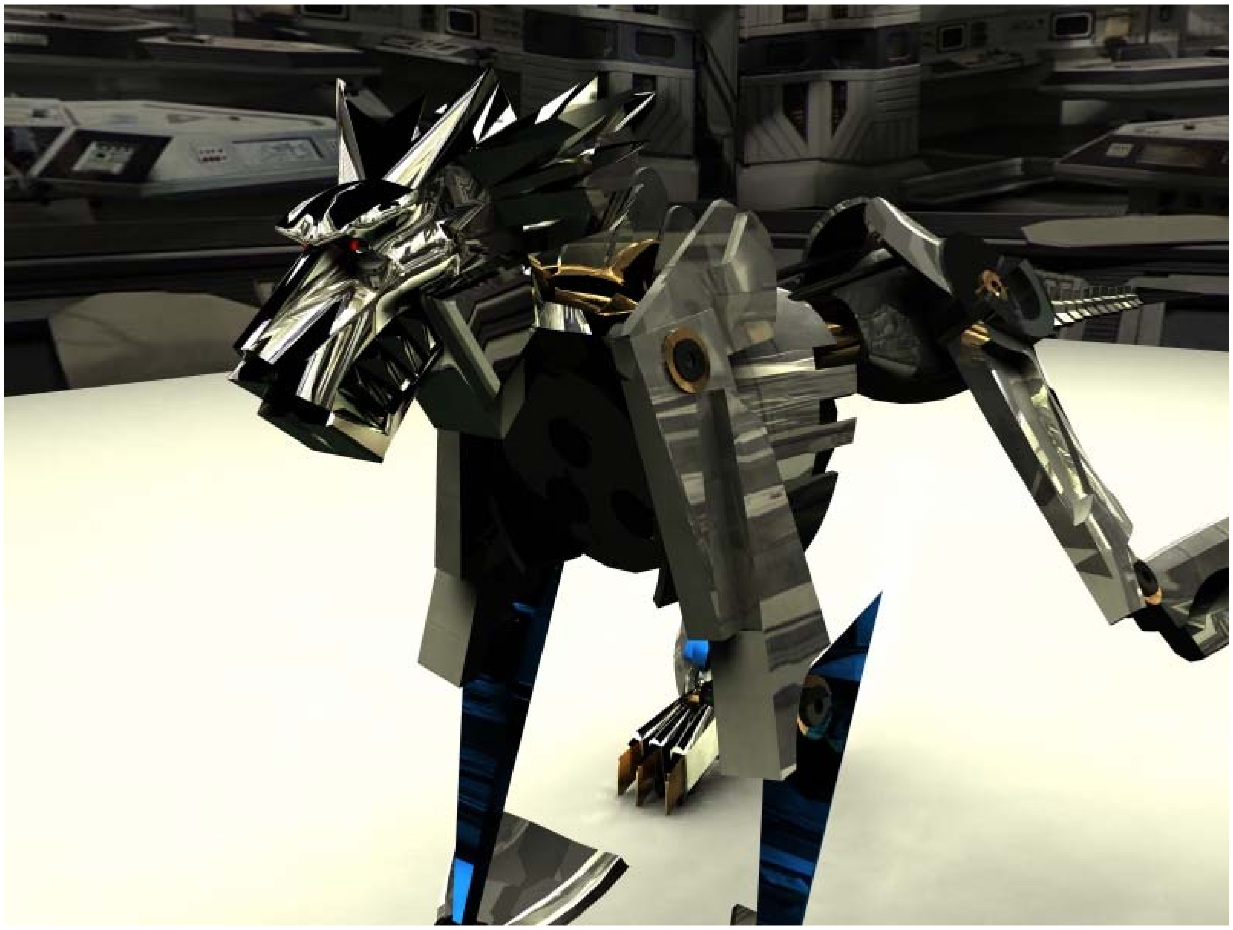


Figure 3.6 An environmentally mapped model and scene. (The most basic form of environmental mapping results in a chrome-like appearance.)

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.

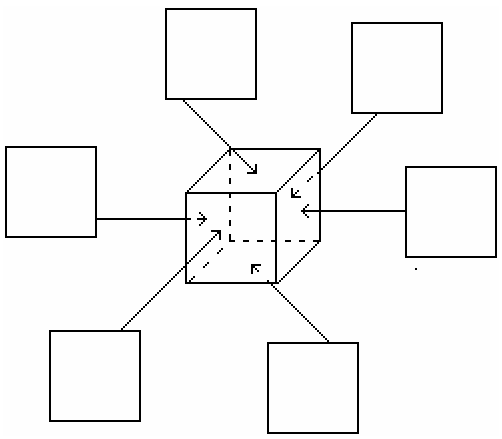
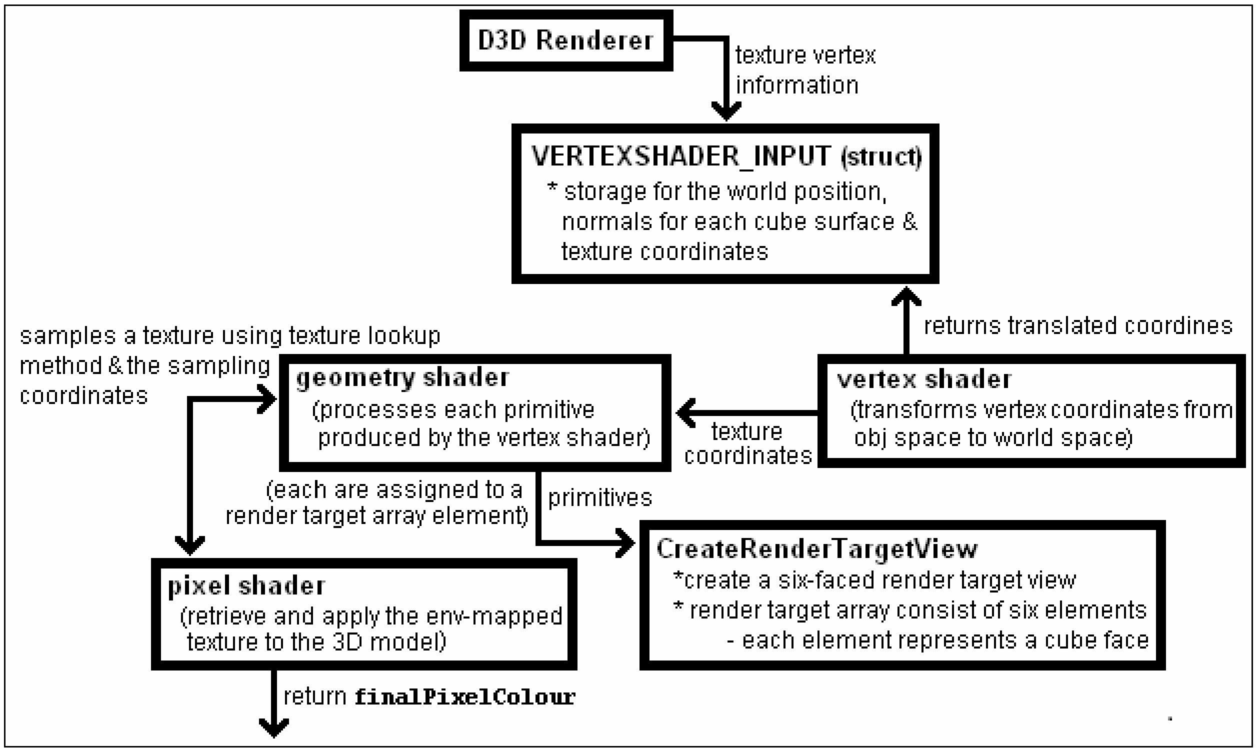


Figure 3.7 A cube map consisting of six texture images.

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


D3D Renderer
texture vertex information
VERTEXSHADER_INPUT (struct)
* storage for the world position, normals for each cube surface & texture coordinates
returns translated coordines
samples a texture using texture lookup method & the sampling coordinates
geometry shader
(processes each primitive produced by the vertex shader)
texture coordinates
vertex shader
(transforms vertex coordinates from obj space to world space)
(each are assigned to a render target array element)
primitives
CreateRenderTargetView
*create a six-faced render target view
* render target array consist of six elements
- each element represents a cube face
pixel shader
(retrieve and apply the env-mapped texture to the 3D model)
return finalPixelColour

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);
}
```

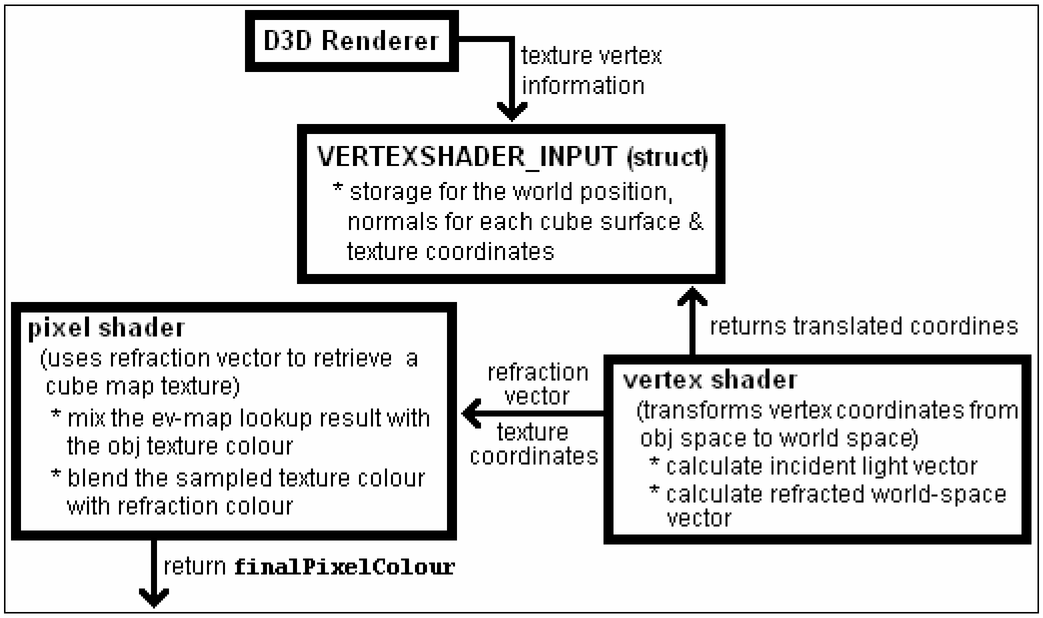
We can now specify the effect technique that will set the previously defined vertex, pixel and geometry shaders. This effect technique has one rendering pass, namely **`P0`**:

```
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


D3D Renderer
texture vertex information
VERTEXSHADER_INPUT (struct)
* storage for the world position, normals for each cube surface & texture coordinates
returns translated coordines
pixel shader
(uses refraction vector to retrieve a cube map texture)
* mix the ev-map lookup result with the obj texture colour
* blend the sampled texture colour with refraction colour
refraction vector
texture coordinates
vertex shader
(transforms vertex coordinates from obj space to world space)
* calculate incident light vector
* calculate refracted world-space vector
return finalPixelColour

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.

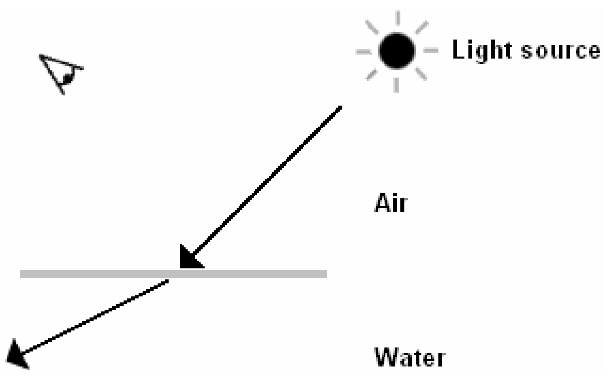


Figure 3.8 Refraction due to light passing from a lower- to a higher density material.

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.

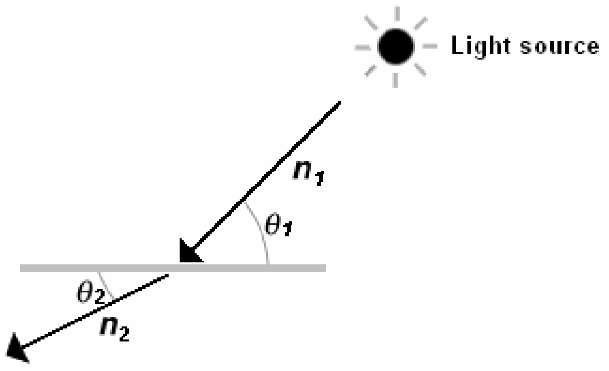


Figure 3.9 Snell's Law

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);
```

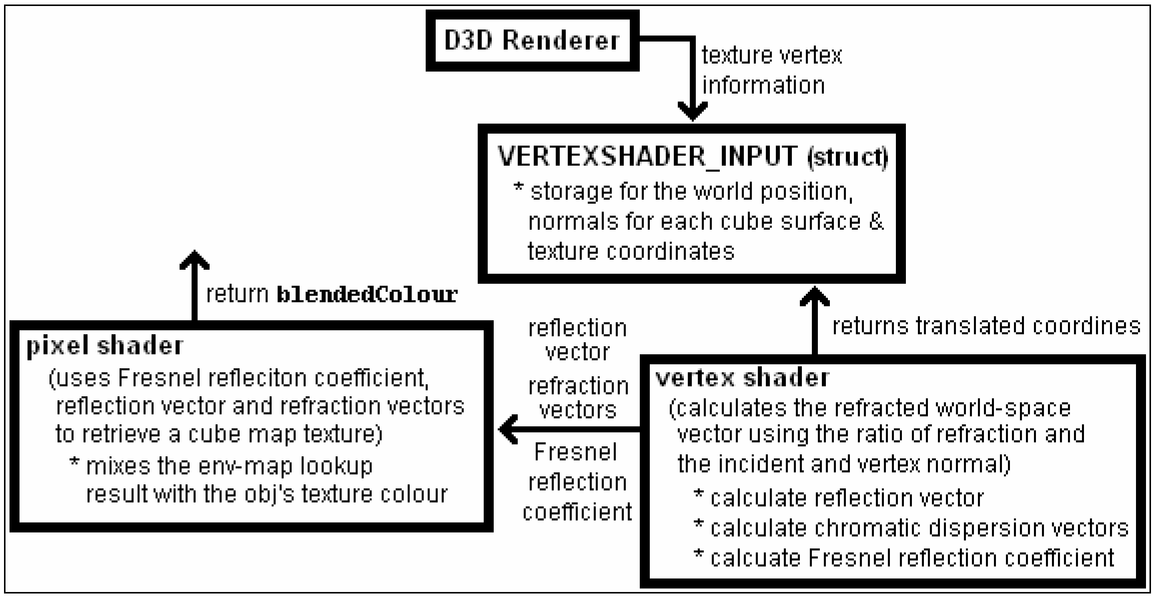
Following this, the sampled texture colour is blended with the refraction colour using the **`lerp`** function. This function performs a linear interpolation, computing the average of two colour samples. Its first two parameters are the colour vectors to average, with its third parameter controlling 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


D3D Renderer
texture vertex information
VERTEXSHADER_INPUT (struct)
* storage for the world position, normals for each cube surface & texture coordinates
return blendedColour
returns translated coordines
pixel shader
(uses Fresnel reflecion coefficient, reflection vector and refraction vectors to retrieve a cube map texture)
* mixes the env-map lookup result with the obj's texture colour
reflection vector
refraction vectors
Fresnel reflection coefficient
vertex shader
(calculates the refracted world-space vector using the ratio of refraction and the incident and vertex normal)
* calculate reflection vector
* calculate chromatic dispersion vectors
* calcuate Fresnel reflection coefficient

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.

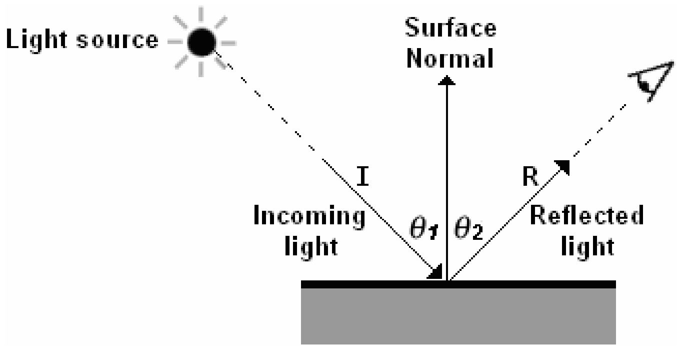


Figure 3.10 Perfect reflection ($\theta_1 = \theta_2$).

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).

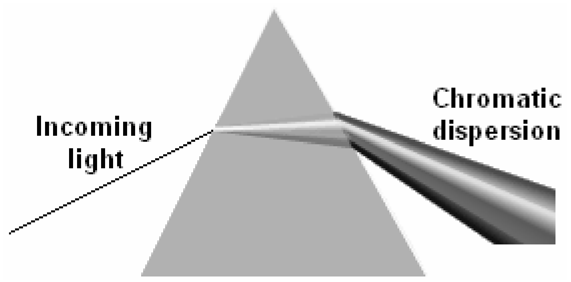


Figure 3.11 Chromatic dispersion of light.

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;
```

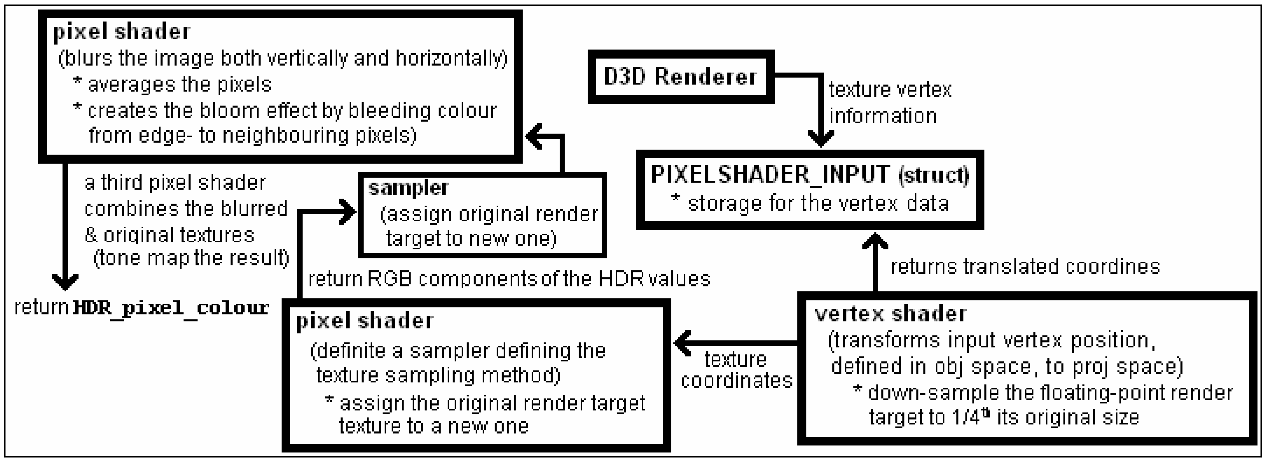
Following this, we blend the sampled refraction texture colour with the reflection colour using the **`lerp`** function. The refraction colour's weight, given as the function's third parameter, is set to the calculated Fresnel reflection coefficient:

```
      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


pixel shader
(blurs the image both vertically and horizontally)
* averages the pixels
* creates the bloom effect by bleeding colour from edge- to neighbouring pixels)
D3D Renderer
texture vertex information
PIXELSHADER_INPUT (struct)
* storage for the vertex data
a third pixel shader combines the blurred & original textures (tone map the result)
sampler
(assign original render target to new one)
return RGB components of the HDR values
returns translated coordines
return HDR_pixel_colour
pixel shader
(definite a sampler defining the texture sampling method)
* assign the original render target texture to a new one
texture coordinates
vertex shader
(transforms input vertex position, defined in obj space, to proj space)
* down-sample the floating-point render target to 1/4th its original size

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.

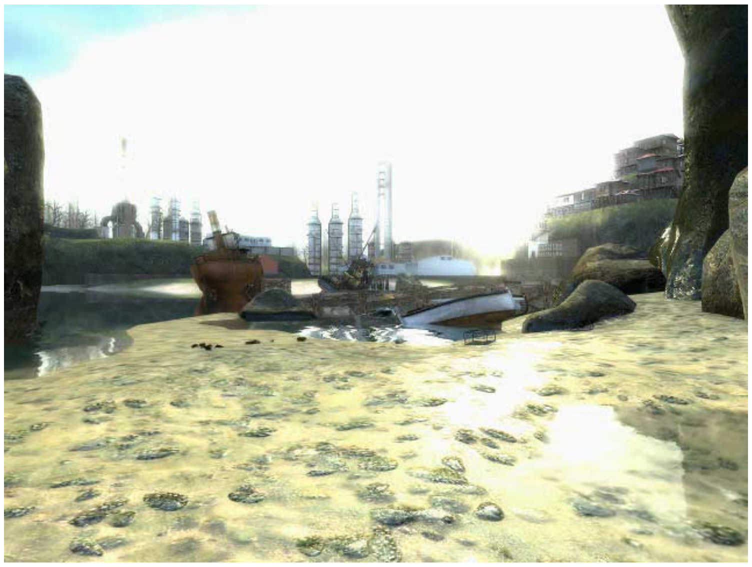


Figure 3.12 High dynamic range lighting.

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.

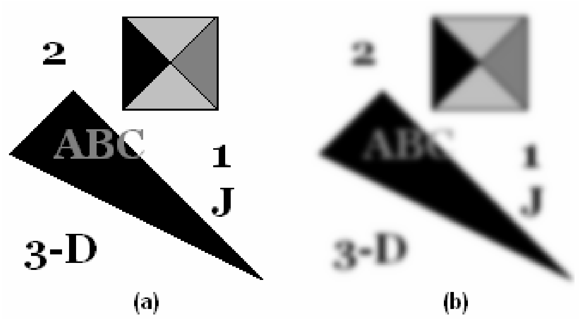


Figure 3.13 (b) Gaussian blur of (a), a simple image.

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.

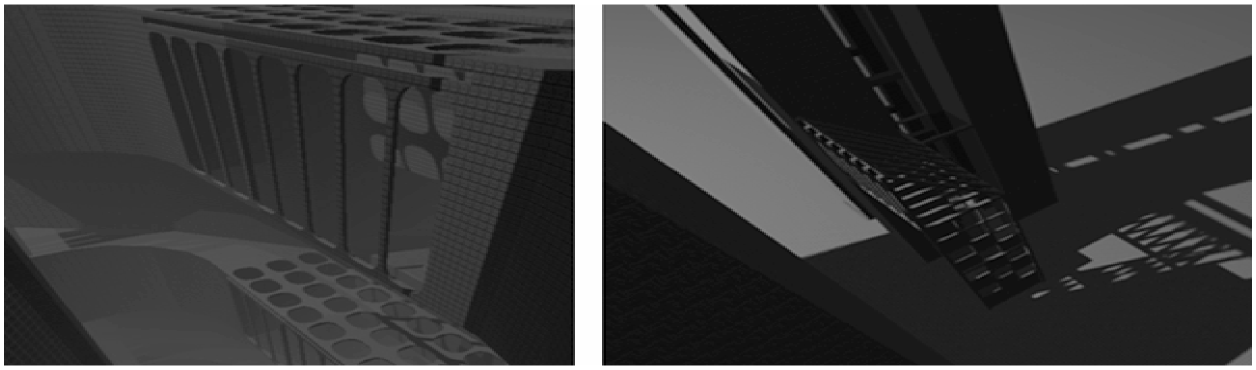


Fig 3.14 Example of stencil shadowing – note the overlapping shadows in the first image.

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.

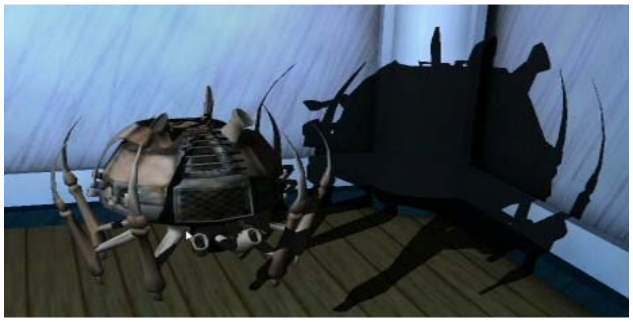


Figure 3.15 Rendering a shadow by means of stencil shadow volumes (using one light source and three-dimensional mesh) – accurately cropped and skewed to fit the surrounding area.

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_STENCILOP_DESC** structure) include the state when stencil testing fails, stencil testing passes and depth testing fails or when both stencil testing and depth testing passes.

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

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

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

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

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

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

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

Table 3.2 Possible stencil operations.

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

```
D3D10_DEPTH_STENCIL_DESC depthstencilDesc;
```

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

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

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

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

```
depthstencilDesc.BackFace.StencilFailOp = D3D10_STENCIL_OP_KEEP;
depthstencilDesc.BackFace.StencilDepthFailOp = D3D10_STENCIL_OP_DECR;
depthstencilDesc.BackFace.StencilPassOp = D3D10_STENCIL_OP_KEEP;
depthstencilDesc.BackFace.StencilFunc = D3D10_COMPARISON_ALWAYS;
depthstencilDesc.FrontFace.StencilFailOp = D3D10_STENCIL_OP_KEEP;
depthstencilDesc.FrontFace.StencilDepthFailOp = D3D10_STENCIL_OP_INCR;
depthstencilDesc.FrontFace.StencilPassOp = D3D10_STENCIL_OP_KEEP;
depthstencilDesc.FrontFace.StencilFunc = D3D10_COMPARISON_ALWAYS;
```

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-interger or signed-interger 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);
```

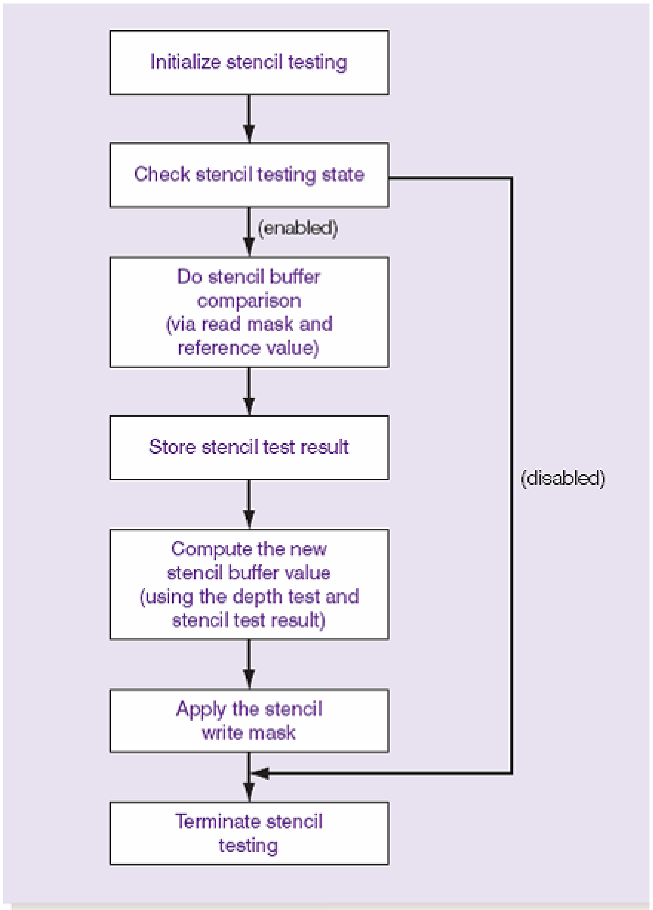
The **`OMSetRenderTargets`** method binds this depth stencil view to the output-merger stage. It takes three parameters, with the first identifying the number of render targets, the second a pointer to a render target view array, and the third a pointer to the to the **`ID3D10DepthStencilView`** interface.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

```
ID3D10RenderTargetView* pRenderTargetView;

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

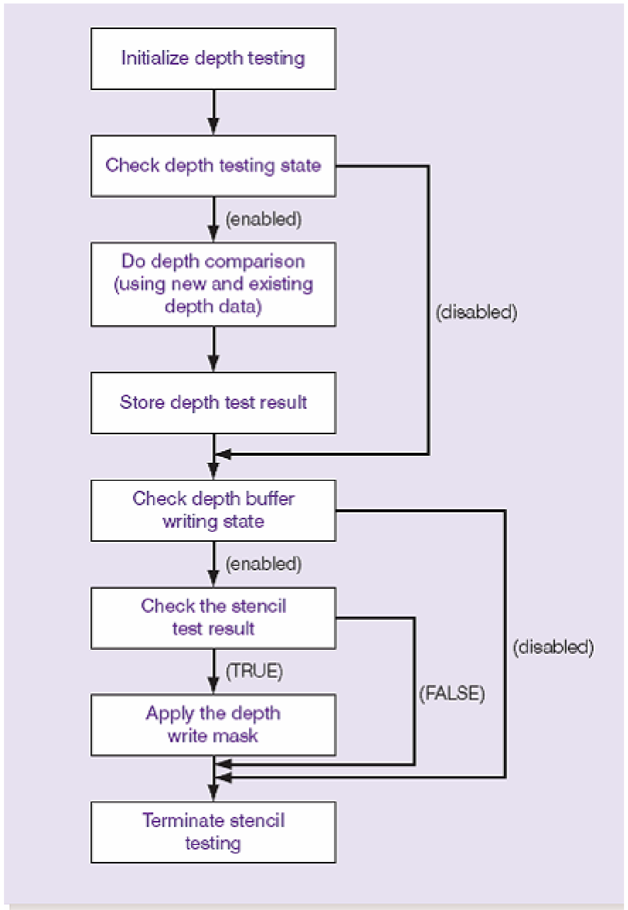
The **`OMSetDepthStencilState ID3D10Device`** interface is used to update the depth stencil state. This update is performed by setting the output-merger stage's depth-stencil state. The **`OMSetDepthStencilState`** method takes two parameters with the first parameter a pointer to an **`ID3D10DepthStencilState`**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



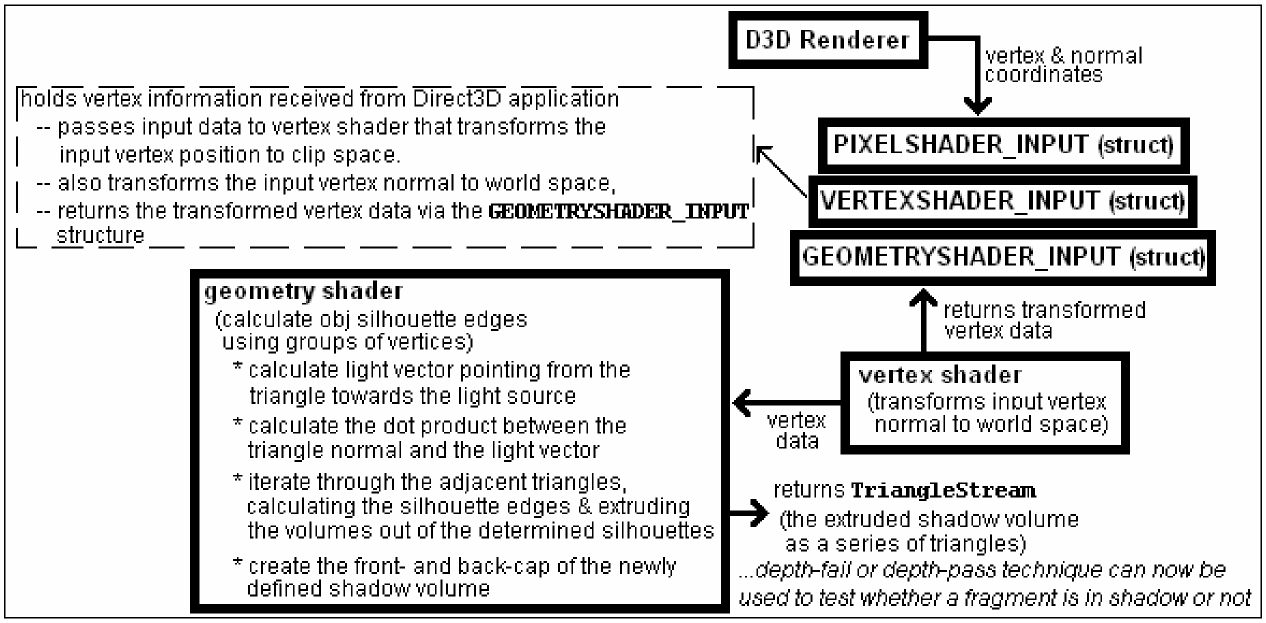
Initialize stencil testing
Check stencil testing state
(enabled)
Do stencil buffer comparison (via read mask and reference value)
Store stencil test result
Compute the new stencil buffer value (using the depth test and stencil test result)
Apply the stencil write mask
Terminate stencil testing
(disabled)



Initialize depth testing
Check depth testing state
(enabled)
Do depth comparison (using new and existing depth data)
Store depth test result
(disabled)
Check depth buffer writing state
(enabled)
Check the stencil test result
(TRUE)
Apply the depth write mask
(FALSE)
(disabled)
Terminate stencil testing

Figure 3.16 The stencil testing process, Figure 3.17 The depth testing process.

Implementing Stencil Shadow Volumes



D3D Renderer
vertex & normal coordinates
PIXELSHADER_INPUT (struct)
VERTEXSHADER_INPUT (struct)
GEOMETRYSHADER_INPUT (struct)
holds vertex information received from Direct3D application
-- passes input data to vertex shader that transforms the input vertex position to clip space.
-- also transforms the input vertex normal to world space,
-- returns the transformed vertex data via the GEOMETRYSHADER_INPUT structure
returns transformed vertex data
vertex shader (transforms input vertex normal to world space)
vertex data
geometry shader (calculate obj silhouette edges using groups of vertices)
* calculate light vector pointing from the triangle towards the light source
* calculate the dot product between the triangle normal and the light vector
* iterate through the adjacent triangles, calculating the silhouette edges & extruding the volumes out of the determined silhouettes
* create the front- and back-cap of the newly defined shadow volume
returns TriangleStream (the extruded shadow volume as a series of triangles)
...depth-fail or depth-pass technique can now be used to test whether a fragment is in shadow or not

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:

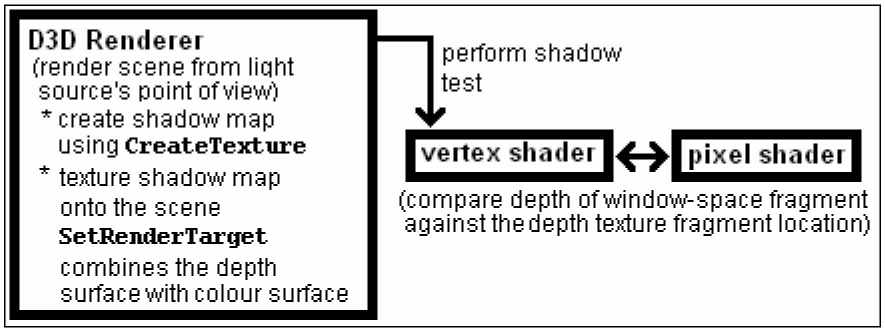
```
[maxvertexcount(18)]
void SilhouetteEdgeAndVolumeGS(triangleadj GEOMETRYSHADER_INPUT vertex[6],
                               float3 normal,
                               inout TriangleStream<PIXELSHADER_INPUT> ExtrudedVolume)
{
     /* determine the light vector from the triangle to light source */
     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


D3D Renderer
(render scene from light source's point of view)
* create shadow map using CreateTexture
* texture shadow map onto the scene
SetRenderTarget combines the depth surface with colour surface
perform shadow test
vertex shader
pixel shader
(compare depth of window-space fragment against the depth texture fragment location)

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

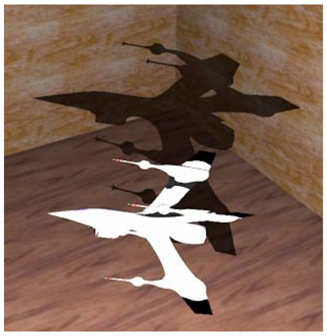


Figure 3.18 Rendering a shadow by means of a shadow map (via one light source and three-dimensional mesh) – accurately cropped and skewed to fit the surrounding area.

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.

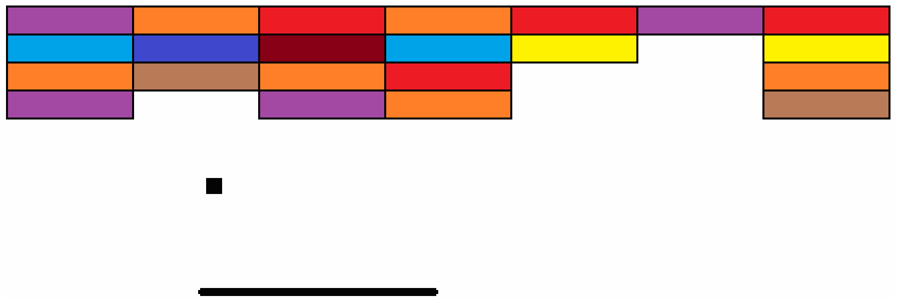


Figure 3.20 A Breakout clone (source code available on the accompanying CD).

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 PPUs. 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 aquiring 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.

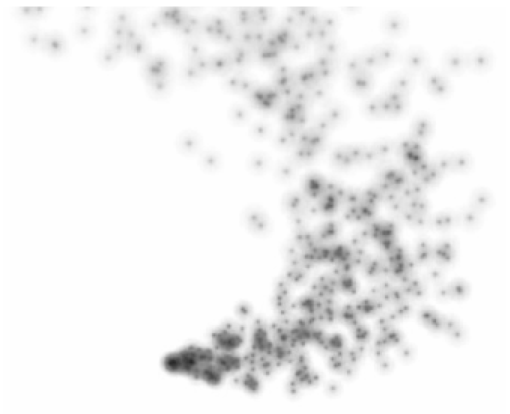


Figure 3.22 Rendering trails with a particle system.

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.

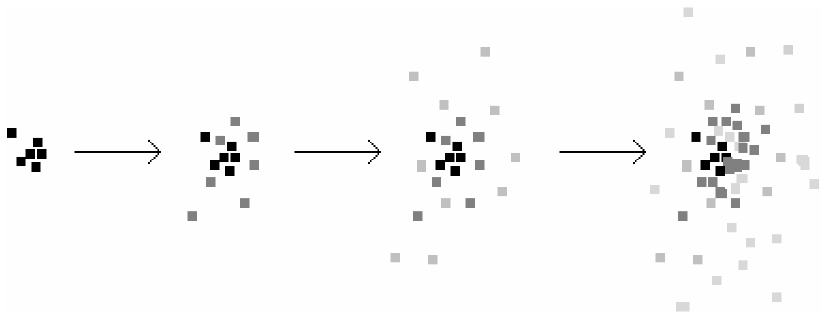


Figure 3.23 Particles being generated over time.

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

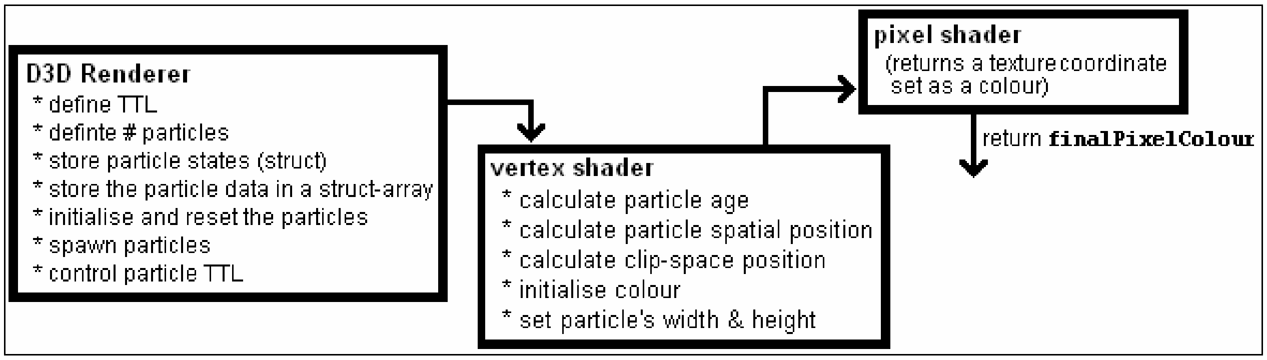
$$V_{new} = V_{old} + g \times t$$

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

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


D3D Renderer
* define TTL
* definte # particles
* store particle states (struct)
* store the particle data in a struct-array
* initialise and reset the particles
* spawn particles
* control particle TTL
vertex shader
* calculate particle age
* calculate particle spatial position
* calculate clip-space position
* initialise colour
* set particle's width & height
pixel shader
(returns a texture coordinate set as a colour)
return finalPixelColour

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.