Coventry University

MASTER OF SCIENCE BY RESEARCH

No fire without smoke: smoke rendering and light interaction for real-time computer graphics

Bass, Christopher

Award date:
2010

Awarding institution:
Coventry University

No Fire Without Smoke;

Smoke Rendering and Light Interaction for Real-Time Computer Graphics

Christopher J. Bass

A thesis submitted in partial fulfilment of the requirements of Coventry University for the degree of

MSc by Research

Submitted: November 2010

No Fire Without Smoke;

Smoke Rendering and Light Interaction for Real-Time Computer Graphics

Christopher J. Bass

Abstract

Realism in computer graphics depends upon digitally representing what we see in the world with careful attention to detail, which usually requires a high degree of complexity in modelling the scene. With some computer graphics applications developers have to limit the complexity of the scene to allow the application to run in real-time on modern consumer grade graphics hardware. This trade-off between realism and performance means that new techniques are continually being developed, the aim of which is to improve the realism of a scene without compromising the real-time performance.

Shadows provide an important visual clue to where objects are positioned in a computer-generated scene, because without shadowing objects can look like they are floating above the ground. Rendering shadows becomes even more important when translucent media, such as smoke, is rendered. Without shadowing smoke can look like it is not part of a scene and without position or shape, appearing as unrealistic.

This thesis charts our investigation of methods for modelling computer generated smoke, techniques for global lighting, and the shadowing of solid objects. We examine the rendering of smoke, its interaction with lighting, and how visually plausible shadows can be added to smoke.

We describe and discuss a novel method for casting shadows through a GPU-based particle system from an opaque solid object. We also describe a new approach for ambient lighting, which we refer to as Surface-Curvature Ambient Occlusion (SCAO), taking advantage of the geometry shader in the Direct3D 10 programmable graphics pipeline.

Table of Contents

Chapter 1 Introduction....9
1.1 Aims....10
1.2 Contribution....10
1.3 Thesis Overview....11
Chapter 2 Programmable Graphics Pipeline....12
2.1 Direct3D 10 Programmable Graphics Pipeline....13
2.2 Input Assembler....14
2.3 Vertex Shaders....15
2.4 Geometry Shaders....16
2.5 Stream Out....16
2.6 Rasteriser....16
2.7 Pixel Shader....17
2.8 Output Merger....17
2.9 Multiple Render Targets....19
Chapter 3 Opaque Solid Model Effects....20
3.1 Bi-directional Reflectance Distribution Function and the Rendering Equation....20
3.2 Lighting Models....20
3.2.1 Lambert....21
3.2.2 Phong and Blinn-Phong....21
3.2.3 Cook-Torrance....21
3.2.4 Oren-Nayar....22
3.2.5 Strauss....22
3.2.6 Ward....23
3.2.7 Ashikhmin-Shirley....23
3.3 Ambient Occlusion....23
3.4 Surface-Curvature Ambient Occlusion....24
3.5 Fresnel Highlighting Effect....28
3.6 Shadow Volumes....30
3.7 Shadow Maps....33
3.8 Smoothies and Penumbra Wedges....38
Chapter 4 Making Smoke....40

4.1 Simulation....40
4.1.1 Grid-Based Eulerian Approach....40
4.1.2 Particle System-Based Lagrangian Approach....41
4.1.3 Smoothed Particle Hydrodynamics....42
4.1.4 Hybrid Methods....43
4.2 Rendering....44
4.2.1 Volume Rendering....44
4.2.2 Particle System Rendering....44
4.2.3 Particle Blending and Sorting....45
4.2.4 Overdraw in Particle System Rendering....46
4.2.5 Soft Particles....47
4.2.6 SPH Rendering....49
Chapter 5 Homemade Smoke and Shadows....51
5.1 Particle System Development....51
5.2 Adding Shadows....56
5.3 Improving our Method by Incorporating Existing Techniques....62
Chapter 6 Final Conclusions....66
6.1 Summary of Contributions....66
6.1.1 External Shadowing of Particle Systems....66
6.1.2 Surface-Curvature Ambient Occlusion....66
6.2 Discussion....66
6.3 Future Work....67
6.3.1 External Shadowing of Particle Systems....67
6.3.2 Surface-Curvature Ambient Occlusion....68
Appendix A....69
Glossary....72
References....74

List of Figures

Figure 2.1.a: Paths through the Direct3D pipeline....13
Figure 2.2.a: Input assembler primitive topologies....14
Figure 2.3.a: 3D coordinate transformations....15
Figure 3.4.a: Ambient estimate comparison....26
Figure 3.4.b: Flowchart detailing the SCAO....27
Figure 3.5.a: The Fresnel effect....28
Figure 3.5.b: The vectors used in the Fresnel highlight effect....29
Figure 3.5.c: The Fresnel highlight effect in different highlight colours....30
Figure 3.6.a: Flowchart detailing GPU shadow volumes in Direct3D 10....31
Figure 3.7.a: Shadow mapping....33
Figure 3.7.b: A basic shadow map with jagged edge aliasing artefacts....34
Figure 3.7.c: Penumbra illustration....36
Figure 4.2.5.a: The soft particles technique....48
Figure 4.2.5.b: The soft volumetric particles technique....49
Figure 5.1.a: Particle system with particles following splines to form a sphere....53
Figure 5.1.b: Particle systems forming smoke plume shapes....54
Figure 5.1.c: Particle systems forming smoke plume shapes with texture....54
Figure 5.2.a: Opacity shadow maps and deep opacity shadow maps....57
Figure 5.2.b: GPU particle system without external shadowing effect....58
Figure 5.2.c: GPU particle system with external shadowing effect....59
Figure 5.2.d: Different view showing our external shadow implementation....60
Figure 5.2.e: Flowchart of the processes in rendering a frame of our model....61
Figure 5.3.a: Soft particles for our technique....63
Figure 5.3.b: Our soft particles....64

List of Equations

Equation 2.8.a: The customisable blending equation.....18
Equation 2.8.b: Additive blending.....18
Equation 2.8.c: Additive alpha blending.....18
Equation 2.8.d: Alpha blending.....19
Equation 3.5.a: Fresnel highlight.....30

Chapter 1 Introduction

A successful modern computer game depends on captivating the user in a convincing and challenging virtual world. Accurate lighting is essential for creating convincing 3D virtual worlds that immerse the user. Real-time speeds are also essential so that the player can interact with the game without any delays, which would break the player's immersion. There are a variety of well-known existing lighting algorithms that are capable of attaining real-time speeds. Many are designed for the lighting of opaque solid objects, but when we want to render matter such as smoke, which is translucent, these lighting models become inadequate. Special effects such as particle systems are often used to create a representation for these substances.

Realistic rendering of natural phenomena such as smoke can help to create a rich virtual world to immerse the user. With the rendering of smoke and translucent gases there are a range of new problems to tackle that do not occur (or are negligible) when rendering solid models. When light interacts with a volume of smoke particles it might pass through, be absorbed or scattered. This means that the rendering of smoke should be dealt with in a different way to rendering solid models to give a realistic output. A significant issue with getting a realistic looking output from a smoke model is the shadowing. There has already been research in this area that focuses on internal shadows and self-shadowing where the smoke casts shadows onto itself and solid objects in the environment. There is not, however, any research that we are aware of looking at external shadows cast onto a volume of smoke particles, which is where this work finds an original topic to investigate.

Developments in computer graphics hardware and 3D graphics APIs (application programming interfaces) have freed software developers from using the fixed function graphics pipeline. With the arrival of the programmable pipeline, developers can utilise the GPU in new ways, allowing new effects to be created to improve the graphics in applications such as computer games.

Adding to the graphical realism in games and virtual worlds adds to the immersion of the user and the overall experience. New and more accurate lighting models than previously possible can be implemented through shaders and used for virtual environments.

For real-time applications such as computer games there is always a trade-off between rendering quality and performance. Rendering smoke with accurate lighting, including scattering and shadowing is a challenge. A sizeable body of research exists which

provides solutions for related topics in this area. Some of these solutions operate at real-time speeds and some are offline techniques. Our work looks at producing a real-time solution, which could be implemented within current computer games and therefore most referenced literature will discuss real-time solutions. Existing solutions in the literature will be discussed in more detail throughout this thesis.

1.1 Aims

The primary aim of this research was to develop a real-time model that improves upon the realism of smoke rendering under different lighting conditions. Through a detailed exploration of the subject area we identified a gap in the existing research, being the casting of external shadows onto a virtual dynamic smoke model. This became the focus of our work from which we developed an original solution.

Branching off from the primary aim, other objectives included exploring into related rendering effects that could be implemented through the programmable graphics pipeline via shaders. This led us into exploring the capabilities of the pipeline and development of a novel ambient lighting estimate for opaque solid models.

1.2 Contribution

The main contribution of this thesis is a method for casting external shadows onto a volume of smoke particles. The smoke particles are simulated using a GPU-based particle system and rendering is via a typical billboard (a camera aligned rectangular primitive with an attached texture) approach.

Another novel contribution of this thesis is the Surface-Curvature Ambient Occlusion (SCAO) technique, an ambient lighting estimate for opaque solid models. This provides a beneficial improvement over using a constant term for ambient lighting. It can easily be implemented on top of another lighting model, which could handle the diffuse and specular terms providing a complete lighting model for opaque solid objects.

Most of the explored techniques and all of the developments are for real-time applications and could be easily used in graphics applications such as computer games with little detriment in performance.

1.3 Thesis Overview

This thesis is divided into 6 chapters, which break down as follows:

Following on from this introduction (**Chapter 1)**, **Chapter 2** details the graphics API that we used during development along with the programmable graphics pipeline discussing the programmable parts of the pipeline and its capabilities. **Chapter 3** examines existing lighting techniques and effects, which can be implemented on opaque solid models. This chapter also covers our innovative ambient lighting model; SCAO. In **Chapter 4** we discuss existing techniques for simulating and rendering smoke in virtual worlds. **Chapter 5** discusses our novel approach to adding shadows that can dynamically interact with a particle system. In **Chapter 6** we make a final conclusion and discuss areas for possible extended research and new ideas. **Appendix A** includes a copy of our earlier Eurographics poster publication [Bass and Anderson 2010] which discussed our external shadowing technique for particle systems.

Chapter 2 Programmable Graphics Pipeline

In recent years developments in graphics-hardware have been replacing sections of the fixed function pipeline to be programmable via so-called shaders. As more of the pipeline is becoming programmable developers have more freedom to harness the power of the GPU for a mixture of applications.

Applications that use accelerated graphics usually make use of an API, the two most popular being either Direct3D [Microsoft 2010] or OpenGL (open graphics library) [Khronos Group 2006]. These API's provide a large set of commands and instructions, giving a basic foundation, which graphics applications can build on.

Direct3D is a platform-dependent graphics API, part of DirectX [Microsoft 2010], requiring a Microsoft Windows operating system. At the beginning of this research project Direct3D version 10 [Blythe 2006] was the most recent available version exposing more programmable parts of the graphics pipeline (geometry shaders) and providing more freedom with development.

OpenGL [Khronos Group 2006] is an open standard cross-platform graphics API, which exists on many different platforms. At the beginning of this project OpenGL version 2.1 was the most recent available version, which does not provide as many programmable parts of the pipeline as the Direct3D version at that time.

The reason for choosing Direct3D was that at the start of this project it allowed more of the pipeline to be programmable. At the time of completing our research project and the writing of this thesis new API versions have been released.

This chapter discusses the Direct3D 10 pipeline.

2.1 Direct3D 10 Programmable Graphics Pipeline

The Direct3D 10 pipeline adds programmable geometry shaders to the already existing vertex and pixel shaders. This allows for a range of new effects and developments to be built on this technology.

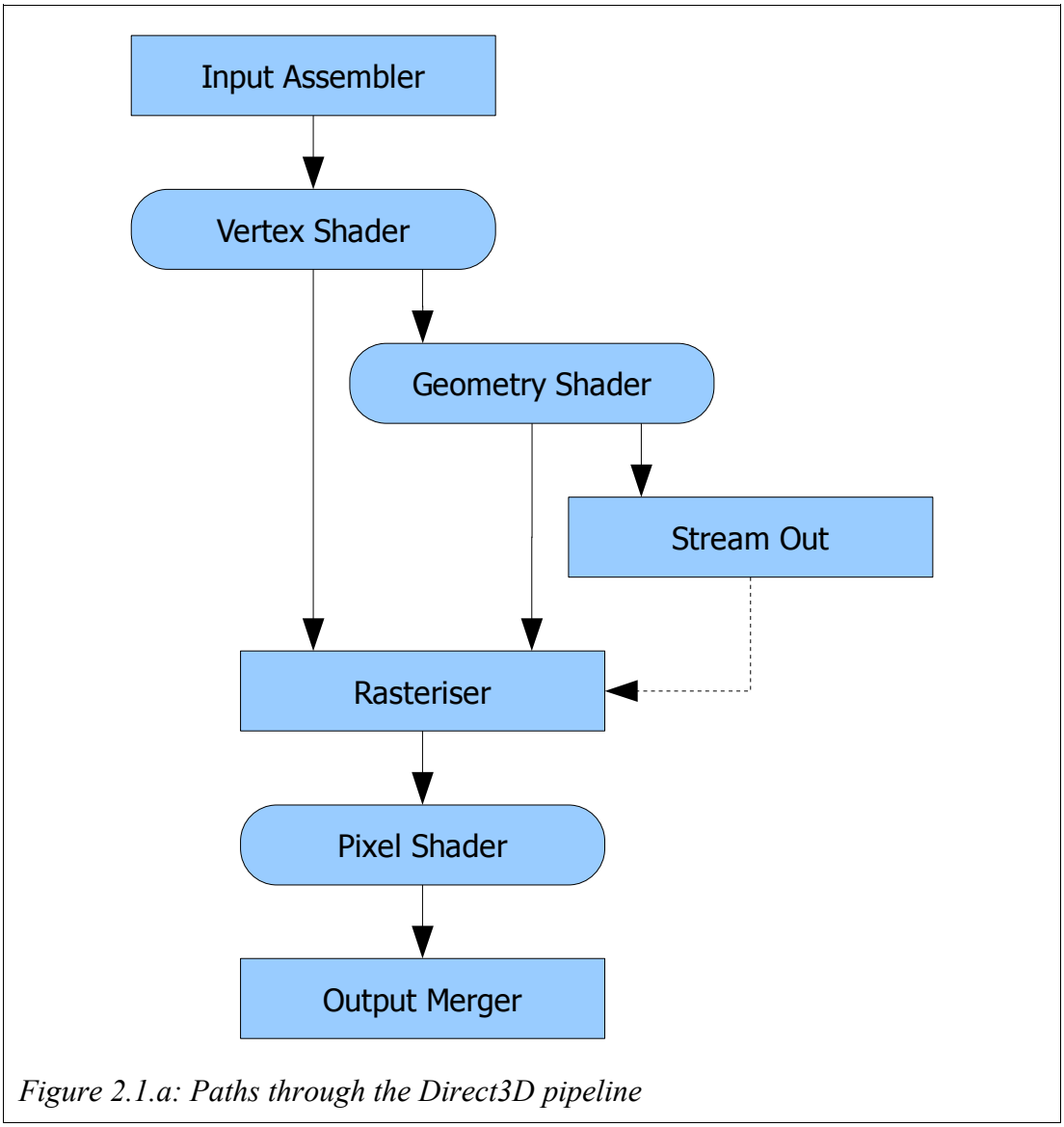

Figure 2.1.a: Paths through the Direct3D pipeline

The Direct3D 10 pipeline can be traversed in a few different ways. The new geometry shader and the stream out stages give 4 possible paths through the Direct3D 10 pipeline as illustrated in Figure 2.1.a. Note that the stream out stage can only be used if the geometry shader is used. Also note that there is the option of passing data to the rasteriser even if the stream out stage is in use. The smooth cornered boxes in Figure 2.1.a identify the programmable stages. All of the stages are discussed in the following sections of this thesis.

2.2 Input Assembler

The purpose of the input assembler is to transfer raw data from memory to the vertex shader. Direct3D 10 provides 9 different topologies for defining the structure that the data is organised in.

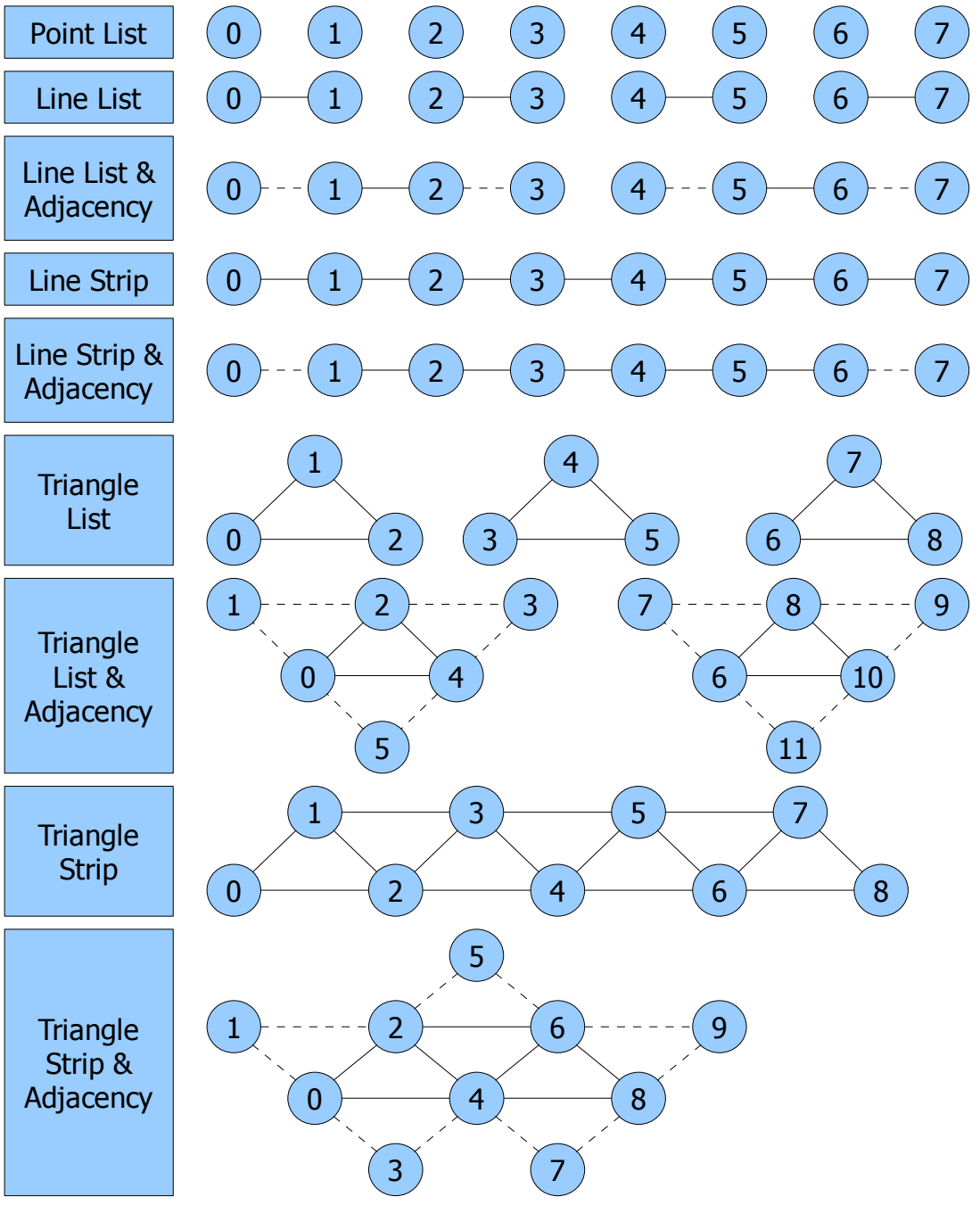

Figure 2.2.a: Input assembler primitive topologies

Figure 2.2.a shows the different topologies available for defining the data structure in Direct3D 10. The point list is a typical choice for particle systems that store single

particle data in each input vertex. It can also be used for drawing single points on the screen. The line list and line strip are for drawing straight lines on the screen with the list used for individual lines and the strip for drawing a joined line. The triangle list and triangle strip are for drawing triangles, the list is for individual triangles and the strip for a joined up strip of triangles. The main difference over the older Direct3D 9 graphics pipeline is the addition of adjacency information, which can be accessed in the geometry shader stage of the pipeline.

2.3 Vertex Shaders

Vertex shaders are the first programmable stage in the graphics pipeline and operate once per incoming vertex, allowing for the vertices to be manipulated in 3D space.

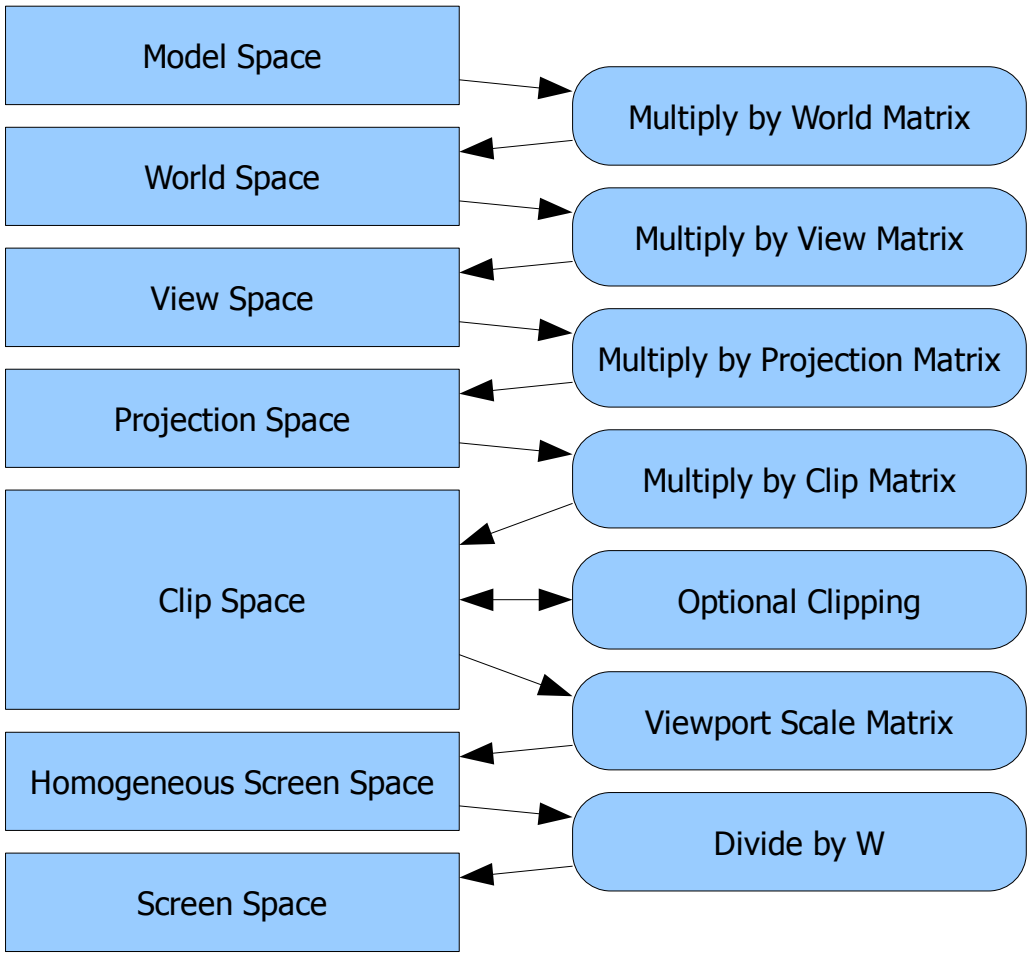

Figure 2.3.a: 3D coordinate transformations

A common task for the vertex shader for rendering 3D meshes from a virtual 3D environment is to convert each vertex from model space to clip space. This is achieved simply by multiplying each vertex by the 'WorldViewProjection' matrix.

2.4 Geometry Shaders

Geometry shaders are a new addition to the Direct3D programmable graphics pipeline introduced with the arrival of Direct3D 10. They are an optional programmable component, which if selected by the developer make up the second programmable stage.

Geometry shaders have similar functionality to vertex shaders but operate on individual 3D primitives instead of individual vertices like the vertex shader does. They also have some other differences to vertex shaders such as access to data from adjacent vertices and the ability to add new primitives or delete existing primitives. These features allow the geometry shader to provide new effects such as the birth and death of particles and creating billboards on the fly from point data. There are other effects that can be created with geometry shaders and there are plenty of examples available in the DirectX SDK [Microsoft 2010], which discuss their capabilities.

Geometry shaders can generate new geometry on the fly by creating and appending new primitives to the output stream. The output stream can be set to a few different topologies, these being a point stream, a line stream, and a triangle stream.

The input and output stream are not required to have the same topology, so for example the input stream could be a point list and the output stream could be a triangle stream, and the geometry shader itself could be turning a single input point into billboards.

2.5 Stream Out

After the geometry shader stage there is the optional stream out function in the Direct3D 10 graphics pipeline, which allows for primitives to be streamed out by writing them back to memory instead of, or as well as, passing them on to the rasteriser. This new operation allows for particle systems to be entirely simulated on the GPU and individual particles can be stored as vertices instead of pixels in a texture.

2.6 Rasteriser

The rasteriser generates the pixels from the incoming vertices, depending on the primitive type. The incoming vertices must have been converted to screen space coordinates by this point; the screen space conversion is usually done inside the vertex or geometry shader. The rasteriser generates pixels depending on the primitive type, so rasterisation of 3 vertices will yield different results depending on the incoming primitive types. The

primitive type *triangle strip* would produce a filled triangle from 3 vertices, whereas the type *line strip* would produce an unfilled triangle, and *point list* would produce 3 separate points. The rasteriser also allows for some geometry to be culled before converting to pixels and passing to the pixel shader.

2.7 Pixel Shader

Pixel shaders are the final programmable stage of the Direct3D 10 graphics pipeline. They operate on each pixel and are often the most performance sapping part of the pipeline due to the large number of pixels passed from the rasteriser, which results in a high number of calculations. Operating on individual pixels means that pixel shaders can produce detailed effects such as per pixel lighting, as opposed to per vertex lighting. Per vertex data such as a normal vector or a colour value are interpolated for pixels depending on the distance from the vertices, so a pixel in-between two vertices would take a weighted average proportion from the normal data and colour data from each. When the lighting calculations are performed in the pixel shader using the interpolated data, the surface appears smooth.

2.8 Output Merger

The final stage of the graphics pipeline is the output merger, which handles the render target and depth stencil buffer. Once a pixel has been shaded the pixel must be compared to what may already be in the render target and then a decision how to overwrite it needs to be made. If depth testing is enabled then a customisable test is performed between the pixel depth and the depth stored in the depth buffer. The result from the depth test can then be used to make a decision on how the pixel should be blended to the render target. For opaque objects it is common to perform a depth test and then overwrite without any blending so that near objects appear in front of objects in the background without any form of blending. Translucent objects require some sort of blending to achieve the desired result. In Direct3D 10 a 'Blend State' is used to determine how pixels are blended when rendering. If left to default the new pixel will overwrite the old pixel (useful for opaque objects).

Porter and Duff [Porter and Duff 1984] discuss the composition of digital images and how the alpha channel can be used to store the opacity of a colour, which can then be used when blending pixels. In the Direct3D 10 programmable graphics pipeline there is

the option of configuring how a new pixel is blended to the render target. The blending equation determines this and it can be customised using Direct3D output merger commands or from HLSL code inside the shader.

Final Colour = (SrcBlend * SrcBlendAlpha) BlendOp (DestBlend * DestBlendAlpha)

Equation 2.8.a: The customisable blending equation

Equation 2.8.a shows the typical format of the blending equation performed in the output merger stage. It can be customised using the available commands. Firstly to use the blending equation blending must be enabled otherwise the output from the pixel shader is used without any blending. With blending enabled the input RGB data sources 1 and 2 (SrcBlend and DestBlend respectively) need to be chosen along with an optional pre-blend operation. The input alpha data sources 1 and 2 (SrcBlendAlpha and DestBlendAlpha respectively) also have options and the optional pre-blend operation is also available. Finally the blending operation (BlendOp) needs to be selected which determines how channel 1 and 2 are combined.

The most commonly used configurations for the blending equation are additive blending, alpha additive blending, and alpha blending.

Final Colour = (SrcColour * One) + (DestColour * One)

Equation 2.8.b: Additive blending

Additive blending is particularly useful for creating transparent objects without an alpha channel. New object colours are added to the existing scene colour. Black (being R=0, G=0, B=0) will act as a transparent colour adding nothing to the scene. Also because addition can be performed in any order, the order of drawing is not important, therefore additive blending is useful for unsorted data.

Final Colour = (SrcColour * SrcAlpha) + (DestColour * One)

Equation 2.8.c: Additive alpha blending

Additive alpha blending is similar to additive blending, but the additional alpha channel on the source allows for any source colour coming from the pixel shader to be made transparent via this alpha value. Note that the colour black will still add nothing to the scene hence will also act as a transparent colour.

$$\text{Final Colour} = (\text{SrcColour} * \text{SrcAlpha}) + (\text{DestColour} * (1-\text{SrcAlpha}))$$

Equation 2.8.d: Alpha blending

Alpha blending is different to additive alpha blending by multiplying the destination colour by the inverse of the source alpha value. This operation means opaque source colours (with an alpha value of 1) will overwrite the destination colour instead of just adding to it as in both additive blending and alpha additive blending. This operation also means that the order of drawing the objects becomes important for correct opacity, therefore objects drawn with alpha blending need to be sorted by depth so that the objects in the distance can be drawn first and the objects nearer to the camera drawn last resulting in correct blending.

2.9 Multiple Render Targets

Multiple render targets (MRTs) allow for the pipeline to output data to different render targets on the same pass. This is a particularly useful tool for outputting additional data as well as colour, such as normals and depth. For example the first pass could render objects in the scene as well as storing additional information in a separate render target, which can then be used in the second pass to create post processing effects or particular lighting effects. An example is the setup described by Fillion and McNaughton [Fillion and McNaughton 2008] who use a setup using 4 MRTs to store information allowing for a range of lighting and special effects to be used in a modern game.

Chapter 3 Opaque Solid Model Effects

With the programmable graphics pipeline there are a range of new special effects that can be used in real-time computer graphics applications such as games. This chapter covers the effects related to opaque solid models including lighting models and shadowing techniques that can be implemented in the Direct3D 10 graphics pipeline. We also lead on to discuss SCAO – our innovative method for improving the ambient lighting term common in some lighting models.

3.1 Bi-directional Reflectance Distribution Function and the Rendering Equation

Bi-directional reflectance distribution function (BRDF) is a function which describes how light is reflected off opaque model surfaces at different incoming and outgoing angles with respect to the surface normal. The Rendering equation is a fundamental concept in the rendering and lighting of many computer generated scenes. It states that the outgoing light leaving a point on a surface is approximately equal to the sum of the emitted light plus the reflected light. This can be written as a function for incoming light and a BRDF. The following Lighting models provide solutions to the rendering equation by assuming a function for incoming light and how the surface reflects light. They can be used to give a reasonably accurate representation for rendering and lighting computer generated scenes.

3.2 Lighting Models

This section discusses the existing lighting models that can be used to shade opaque solid models under direct illumination from a light source. In these models lights are usually modelled either as a directional light or as a point light. Directional lights have direction and colour and can be effective in modelling distant light sources in computer generated scenes. Point lights have a position and colour and can be effective in modelling nearby light sources emitting from a single point. Many of these lighting models are built from separate different forms of lighting, which are calculated as separate terms before being added together as in the Phong and Blinn-Phong lighting models.

3.2.1 Lambert

Lambert reflectance states that the light reflected off a surface will be the same regardless of the viewing angle. This type of reflectance is particularly effective in modelling rough surfaces as it gives objects a matt look. Lambert reflectance is sometimes used for modelling the diffuse reflection term found in some of the following lighting models (see 3.2.2).

3.2.2 Phong and Blinn-Phong

The Phong [Phong 1973] and Blinn-Phong [Blinn 1977] lighting models break down shading into three different elements, these being ambient, diffuse, and specular, which when added together give the complete lighting model.

The ambient component is usually just a constant RGB colour value, which allows the objects in the scene to be visible even if there are no other lights.

The diffuse component is dependent on the angle at which light hits a surface; it is calculated by finding the dot product between the direction of the light source and the direction of the surface normal the diffuse term exhibits Lambert reflectance.

The specular component is dependent on the angle at which light hits a surface and the position of the viewer. It is calculated in slightly different ways by the Phong and Blinn-Phong models. The Phong model finds the dot product between the reflection vector and the viewer. The Blinn-Phong model simplifies the calculation for directional lights by first calculating the half-vector between the light and the viewer direction vectors and then finding the dot product between the half-vector and the surface normal.

3.2.3 Cook-Torrance

The Cook-Torrance lighting model [Cook and Torrance 1982] allows for more accurate modelling of rough surfaces, which is something the Blinn-Phong model does not consider. It is particularly effective for modelling rough metals. Like the Blinn-Phong model, the Cook-Torrance model breaks down into the three lighting elements, ambient, diffuse, and specular, with the main difference being the calculation of the specular term. The surface roughness is modelled based on the micro-facets method, where a rough surface is assumed to have evenly spaced (smaller than per-pixel) facets cut into the surface, which change the specular reflection properties of the modelled surface.

Comparing the Blinn-Phong model to the Cook-Torrance model for the calculation of the specular component, there are three new elements introduced into the equation, these being Fresnel, roughness, and geometric.

The Fresnel term improves the specular reflections by making the intensity of the specular highlight stronger at shallower angles. This occurs in nature due to more light being reflected at shallow angles instead of being non-view-dependent as is the case with the Blinn-Phong model.

The roughness element describes the distribution of the micro facets that face in the same direction as the half-vector. A rougher surface will see a larger specular reflection compared to a smoother surface with a small specular reflection.

The geometric term describes the amount of incoming and reflected light blocked by the micro-facets at shallow angles. With deeper micro-facets more light is blocked and results in rough surfaces appearing duller than smooth surfaces.

These terms can be calculated in a number of different ways, making this a very flexible model, nevertheless this model can take a long time to fine tune to achieve the desired results.

3.2.4 Oren-Nayar

The Oren-Nayar [Oren and Nayar 1994] lighting model focuses on modelling only the diffuse term and the effects of the surface roughness on the diffuse term. Similar to the Cook-Torrance model, the micro-facet concept to modelling rough surfaces also occurs in the Oren-Nayar model. The micro-facets concept is developed to include inter-reflection between facets giving the effect of blurring the light leaving a surface.

In its full evaluation it is one of the most demanding and complex models, which can be too much of a hit on performance for some applications. For this reason there are simplified versions, which trade accuracy for better performance.

3.2.5 Strauss

The Strauss [Strauss 1990] lighting model was developed with users in mind by featuring easy to understand tuning terms with ranges between 0 and 1. This feature means it can be quickly tuned to model a range of smooth and rough, metals and plastics. Although it is a somewhat recent model, it does not bring any novel ideas to already existing models.

3.2.6 Ward

Ward [Ward 1992] developed his model to be an approximation of empirically observed results. It is different to previous models by introducing anisotropic (instead of isotropic, assumed by previous models) specular highlights. Being anisotropic means directionally dependent, so rotating an anisotropic surface around its surface normal will change its reflection properties. This allows Ward to provide a superior model for materials other than metals and plastics. In particular it is well suited to modelling materials with micro-facet patterns such as wood (grain), fabrics, and brushed steel. This model can be tuned to control the direction and strength of the grain to yield different results.

3.2.7 Ashikhmin-Shirley

The Ashikhmin-Shirley [Ashikhmin and Shirley 2000] lighting model takes inspiration from some of the previously discussed models. Inspired by Ward, Ashikhmin-Shirley developed their model to approximate empirically observed results. Like Ward, it is also an anisotropic model, making it well suited to modelling materials with grain.

Similar to the Cook-Torrance model, the Ashikhmin-Shirley model employs a Fresnel weighting element in the specular component to get a more physically accurate reflection.

Physical plausibility of the results was an important factor during the development of this model and by adding the Fresnel term the energy was unbalanced, which lead to their proposition of using a non-Lambertian diffuse component.

Although the Oren-Nayar model also has a non-Lambertian diffuse component, the Oren-Nayar model is not referenced in Ashikhmin-Shirley [Ashikhmin and Shirley 2000] and the calculation of the diffuse term is different.

The Ashikhmin-Shirley model requires a more complex evaluation so may require some performance enhancing optimisations for implementation in some applications. It can produce renders that none of the aforementioned models can match single-handedly, due to being anisotropic and including a non-Lambertian diffuse term.

3.3 Ambient Occlusion

In section 3.2 we discussed some lighting models which can be used to shade individual objects in a scene. These lighting models can provide adequate shading for objects under direct illumination from directional and point light sources. Objects not in direct illumination require global illumination to make them visible. Global illumination is

perhaps the most complex form of lighting that is often simplified the most; as is the case with the Ambient lighting term which is just a single constant! Ambient light is used to model light which has bounced many times and when simplified to just one constant is assumed to have no direction and have a constant intensity, with the purpose to make all the objects in a scene visible even when there are no lights. Offline models such as photon mapping [Jenson 1996] can estimate the path light takes usually for a set maximum number of bounces and can provide detailed renders but the computational workload of these calculations is usually too high for real-time applications. Ambient occlusion is an approximation to global illumination helping to add to the realism of the lighting in the scene. To avoid the calculation-intense method of finding the full path which light takes from the light source to the surface. The ambient occlusion model usually works backwards by casting out single rays from the surface into the scene to determine if there are any occluding surfaces nearby which would reduce the amount of ambient light reaching the surface.

3.4 Surface-Curvature Ambient Occlusion

With the programmable graphics pipeline new real-time models have been developed and implemented to give a better estimate for the ambient term. SSAO (screen-space ambient occlusion) [Mittring 2007] is a technique for modelling global illumination in real-time by estimating the ambient occlusion in the screen space. The estimation is performed by comparing the depth values in the scene depth buffer to the local values selected via a randomly rotated kernel giving a variance which can be used to shade the pixels according to high frequency changes where deep variance results in a darker shading. This technique has advantages being independent from the scene complexity, easy-to-integrate into a modern graphics pipeline using the existing data in the depth buffer, and can be processed using the GPU. It has disadvantages being view-dependant, adding noise, and bleeding at depth discontinuities near object edges.

Filion and McNaughton [Fillion and McNaughton 2008] present another version of SSAO with a similar approach. Between 8 and 32 samples are taken around the pixel position using a texture full of random vectors to determine which nearby pixels are selected. The depth values are compared and if the pixel depth is greater than the sample's depth then that sample contributes to the occlusion. With a greater difference between the depths the occlusion factor is greater.

Bavoil and Sainz [Bavoil and Sainz 2008] present another different version of SSAO involving tracing rays in 2D across the depth values in screen space. There is a demonstration program of this model in the NVidia Direct3D SDK [NVidia 2007].

Pharr and Green [Pharr and Green 2004] present an older ambient occlusion model that makes an estimate of how much ambient light can reach a particular part of a surface. They use an offline technique involving a ray tracer to build an ambient occlusion map of a model that can be used during rendering to give a better estimate of the ambient occlusion term.

Bunnell [Bunnell 2005] presents a faster ambient occlusion model that avoids casting rays. Disks are used to approximate the ambient occlusion of objects in a scene. For nearby occluding meshes one disk per vertex can be used but for distant objects this can be simplified further to fewer disks or even a single disk. Then for each vertex, shadowing information from all the disks is summed up to give the approximate ambient occlusion for that vertex. Hoberock and Jia [Hoberock and Jia 2007] build on this ambient occlusion model [Bunnell 2005] by allowing for higher quality, per-pixel accuracy, and reducing the present artefacts.

Figure 3.4.a: Ambient estimate comparison

Inspired by the SSAO models and ambient occlusion literature we developed a model by investigating a method to create an ambient occlusion effect using the geometry shader. The geometry shader has the ability to retrieve the data of adjacent vertices and by taking the normal vectors of them one can calculate the curvature of a surface and determine whether it is convex or concave. Then by making the assumption that less ambient light can reach a concave surface a better ambient lighting estimate can be made for shading the surface. The result is Surface-Curvature Ambient Occlusion (SCAO), an easy-to-implement and efficient shader that visually improves the ambient estimate for the ambient term in a lighting model. Figure 3.4.a shows a comparison between the results of using a constant ambient term (left) versus the result from our novel SCAO technique (right).

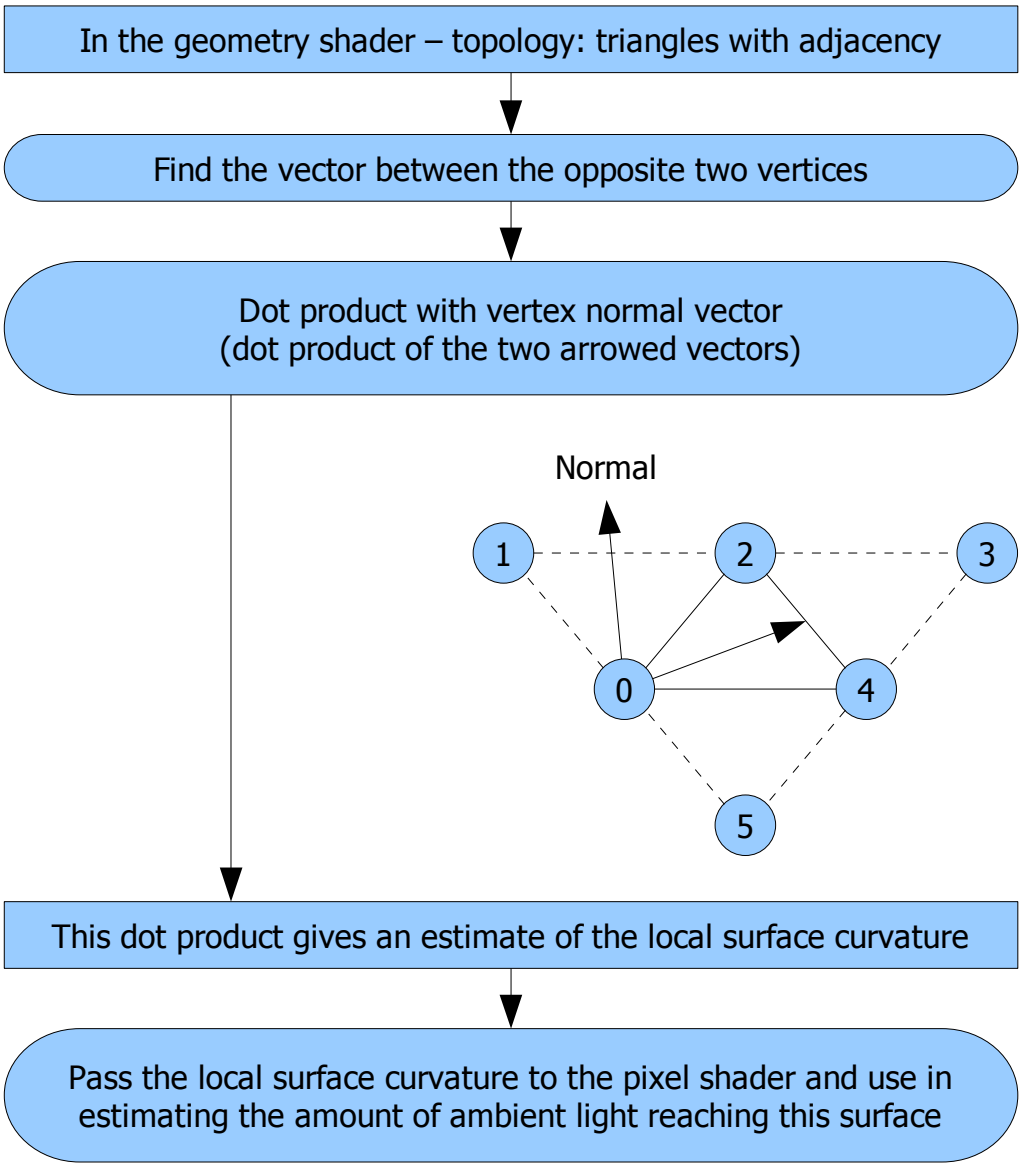

Figure 3.4.b: Flowchart detailing the SCAO

Figure 3.4.b shows the SCAO as it currently stands. It is quite a simple idea and gives a rough estimate of the ambient light reaching a surface but makes a big improvement over just assuming ambient light is the same everywhere (by setting it to a static unchanging constant). Compared to the discussed ambient occlusion models it should provide an improvement on performance, especially to the more complicated SSAO methods. Our method does not require any additional data other than the mesh vertices allowing it to be integrated with any other global illumination lighting models with ease.

Our SCAO shader provides a novel implementation to ambient lighting using the geometry shader to detect the local surface curvature of the model. The dwarf model that this shader was tested on is a low polygon model of roughly 1000 faces. On more

detailed models often found in modern games the polygon count is a lot higher. On a higher polygon model we expect the results to be more accurate resulting in an improved visual output. For future work the adjacent faces could be used in addition to get more samples and an average of the local surface curvature.

3.5 Fresnel Highlighting Effect

The Fresnel effect is the effect where more light reflects off a surface at a shallow angle, and less at a steeper angle. It is a term incorporated in some of the more complex world lighting models, which were discussed in a previous section.

For computer games this effect can be recreated ignoring the light position and with a variety of colours to provide an effective way of highlighting in-game objects. The algorithm works by outputting a highlight colour from the pixel shader when the surface normals face away from the camera. This effect can also be applied to an object that is bump mapped for a rough bumpy highlight effect around the edges of an object.

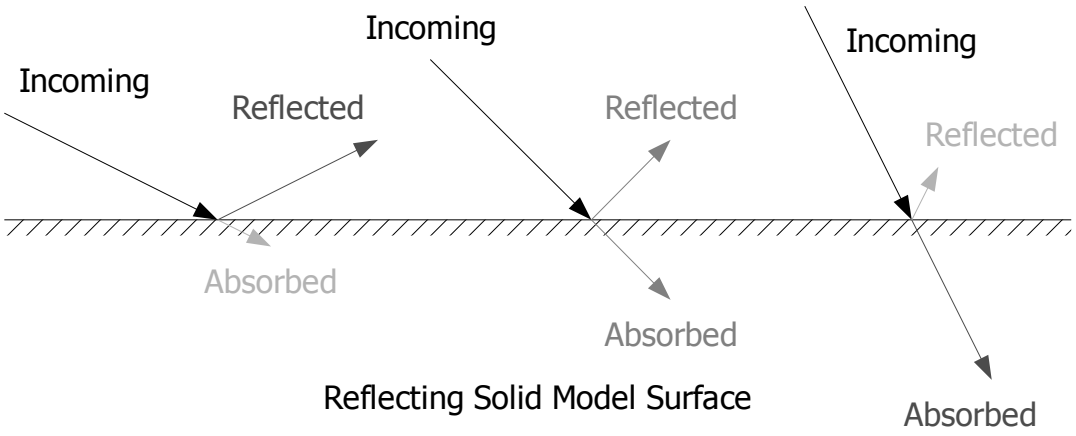

Figure 3.5.a: The Fresnel effect

Figure 3.5.a shows the different amounts of light reflected at different angles.

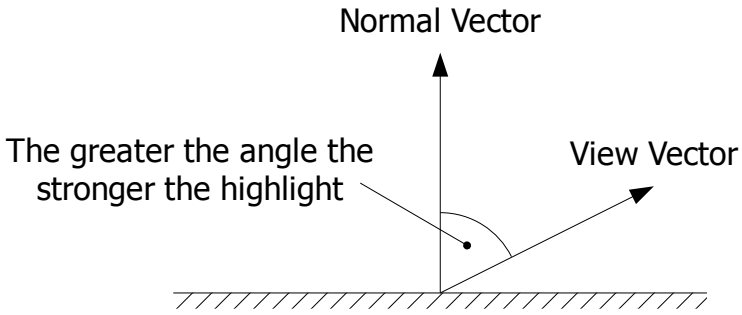

Figure 3.5.b: The vectors used in the Fresnel highlight effect

Figure 3.4.a and Figure 3.5.b show the relationship between the angle and the amount of reflected light. By applying a highlight colour in the pixel shader that varies depending on the angle between the view and normal vector a Fresnel highlighting effect can be created.

Figure 3.5.c: The Fresnel highlight effect in different highlight colours

Figure 3.5.c shows the Fresnel highlight effect. Note that tweaking the constant values in Equation 3.5.a will result in a variety of different outputs. The current settings for these constant values and the equation itself are given in Equation 3.5.a.

$$\text{PixelColour} = \text{LightingModelColour} + \text{FresnelHighlightColour}$$

where,

$$\text{FresnelHighlightColour} = \text{HighlightColour} * (0.7 - \text{dot}(\text{ViewVector}, \text{NormalVector}))^4$$

Equation 3.5.a: Fresnel highlight

3.6 Shadow Volumes

Shadow volumes [Crow 1977] are a method for adding shadows to solid objects in a scene. Shadow volumes can be created from the solid object geometry by extruding the back face of the model to infinity (or similar technique). Pixels inside the shadow volume are shaded whereas those outside are not, which is usually determined via a stencil buffer

implementation, such as ‘depth fail’ [Bilodeau and Songy 1999] which is also known as ‘Carmack’s reverse’ who independently discovered and advertised the technique [Carmack 2000]. Everitt and Kilgard [Everitt and Kilgard 2002] give a detailed discussion of the depth fail algorithm. This implementation improved on previous techniques by fixing the errors occurring when the camera was positioned inside the shadow volume.

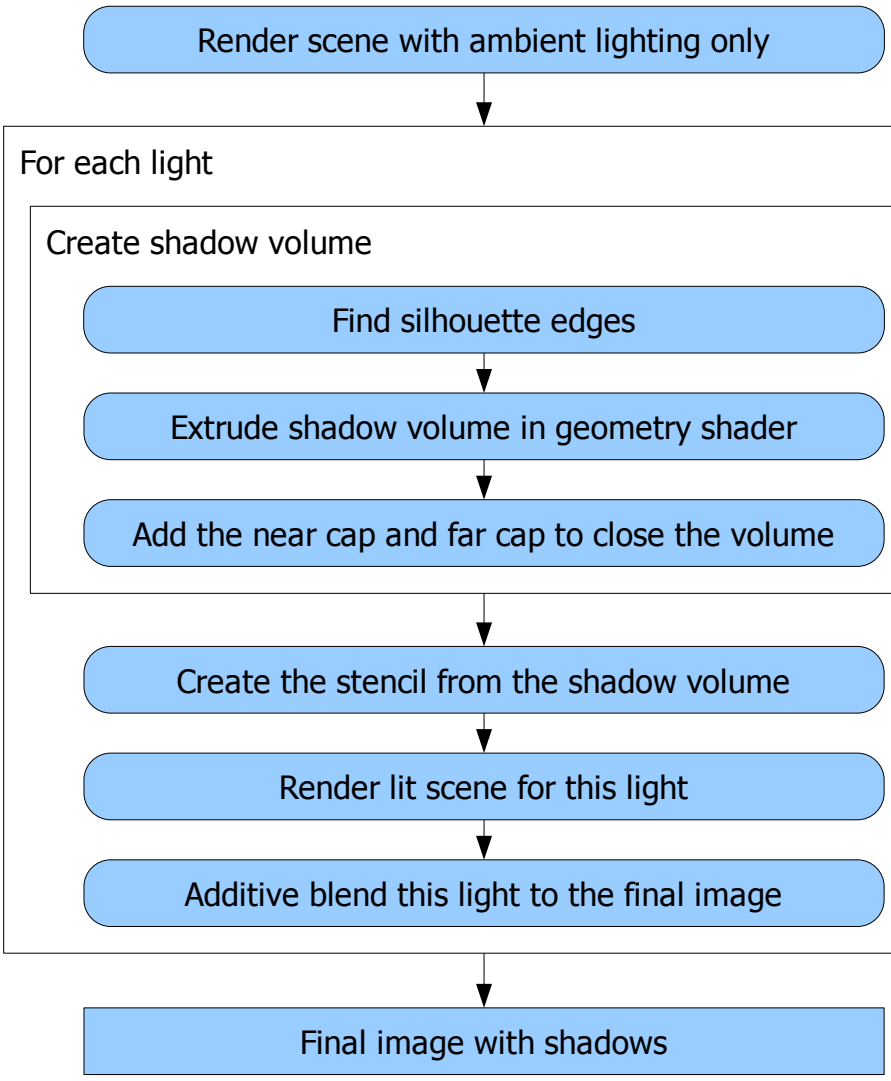

Figure 3.6.a: Flowchart detailing GPU shadow volumes in Direct3D 10

Figure 3.6.a shows a GPU shadow volume implementation which makes use of the geometry shader. The programmable geometry shader in the Direct3D 10 graphics pipeline allows shadow volumes to be extruded and capped (closed) on the GPU improving the real-time performance of this shadowing technique over earlier implementations. An implementation using the GPU to extrude the shadow volume can

be found in the DirectX SDK [Microsoft 2010] (and SDK versions from later than December 2005) under 'ShadowVolume10'.

The geometry shader can be used to find the silhouette edges by looking at the normals of the two adjoining faces (by using an input topology of triangles with adjacency), if one faces towards the light source and the other faces away from the light source then the edge is a silhouette edge. Once the edge has been found it can be extruded away from the light direction and capped to form the shadow volume. Front capping can just use the front faces as they are, back capping can be achieved by using the front faces translated a distance away from the light direction at each vertex.

By building the shadow volume a two sided stencil can be rendered which masks the shadowed parts of the scene. It operates by comparing the depth from the ambient pass to the back faces and incrementing the stencil, then comparing to the front faces and decrementing the stencil. This results in a stencil where the non-zero values represent pixels inside the shadow volume. Lighting for that light source can then be additively blended to each pixel in the scene where the stencil value is equal to zero.

Shadow volumes offer a real-time solution to shadowing solid objects which results in fairly accurate shadows for point lights with hard edged shadows without a penumbra. A more advanced shadow volume algorithm is needed to add soft edged shadows to represent area lights. Assarsson *et al.* [Assarsson et al. 2003] discuss a soft shadow volume technique which they claim to be real-time, however a shadow mapping technique usually runs faster especially on large scene with lots of geometry. Donnelly and Demers [Donnelly and Demers 2004] noted that the method discussed by Assarsson *et al.* would not result in a real-time solution in their high geometry scene, instead they proposed a static lighting solution which could produce soft shadows for static geometry where the light source moved in a consistent pattern.

Another difficulty for shadow volumes is that they are usually restricted to closed meshes, making them unsuitable in other situations. Stich *et al.* [Stich et al. 2007] present a robust shadow volume technique that can handle open meshes by extruding individual triangles. This could be useful for shadowing point sprite particles, however this approach is computationally demanding which could be an issue for real-time applications.

The shadow volume technique has an artefact caused by the limiting granularity of the edges between faces when calculating the silhouette edge. The silhouette is built from

vertices which can result in an irregular sawtooth silhouette edge. On low-polygon meshes this artefact is pronounced whereas it is not as noticeable on higher-polygon meshes. A possible solution is to use a depth bias which would offset the shadow volume a little, so as to avoid some of the artefact's evident self-shadows.

Often a better solution is to use shadow maps, which do not suffer from this artefact and tend to be less costly on large scenes. We will discuss shadow maps in the following section.

3.7 Shadow Maps

Shadow maps [Williams 1978] provide an alternative method for adding shadows to a scene. For this, first a depth map is generated from the light source point of view, mapping the distance from the light source to all the occluding shadow objects. Then, during the rendering of the final scene, a comparison is made between the distance of the pixel to the light and the value that is stored in this height map. If the distance to the light is greater than the value stored in the height map then the pixel is in shadow and rendered accordingly.

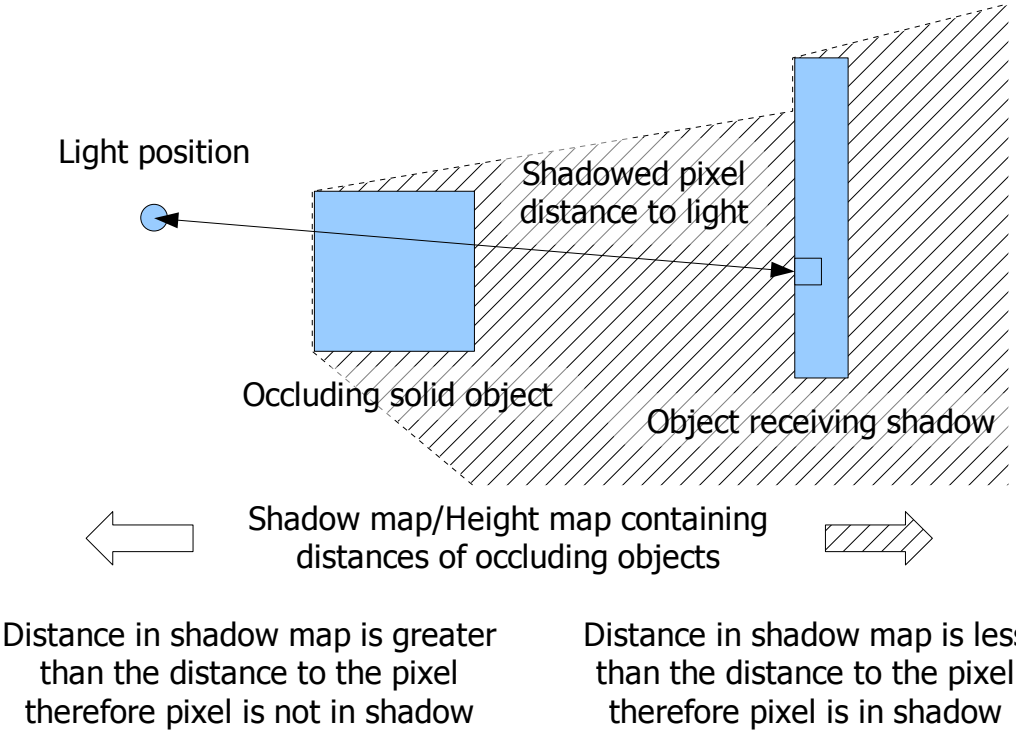

Figure 3.7.a: Shadow mapping

Figure 3.7.a illustrates how the basic shadow mapping algorithm works.

Figure 3.7.b: A basic shadow map with jagged edge aliasing artefacts

Figure 3.7.b shows a common artefact of the basic shadow map implementation [Williams 1978], in which the pixelated edges of the shadow are caused by the limited granularity of the shadow map texture. Fortunately there exists a large number of published advanced shadow mapping techniques for reducing or removing these aliasing artefacts that produce smooth edges to a shadow representing the penumbra caused by area lights. Another artefact also caused by the limited granularity of a shadow map is that shadow maps can miss high-frequency shadows where the shadow caster is small enough to be in-between two samples of the shadow map. We can see the effect of this in Figure 3.7.b at the narrowest and sharpest points on the sword hilt where the shadow map granularity cannot capture all of the fine detail because it is sampling at a lower frequency than the detail.

Reeves *et al.* [Reeves et al. 1987] presented the original percentage closer filtering (PCF) technique for smoothing the edges of a shadow created by shadow mapping. Bunnell and Pellacini [Bunnell and Pellacini 2004] describe a more recent adaptation version of PCF in an article in *GPU Gems*. They adapt the original technique so that an implementation on modern hardware (through pixel shaders) is straightforward and more efficient. The PCF technique involves taking multiple samples from the depth map, determining if they are in shadow or not, and then taking an average to build a 'shadow coefficient'. The 'shadow coefficient' then gives a gradual drop off at the shadow edge, resulting in a smoother edge with less noticeable aliasing artefacts.

Donnelly and Lauritzen [Donnelly and Lauritzen 2006] present variance shadow maps (VSM), which provide a solution to the aliasing artefacts in standard shadow maps.

Instead of storing just the depth when rendering from the point-of-view of the light source (as in original shadow map implementations) they calculate and store the mean and the mean squared of a distribution of depths. This allows for the variance to be calculated and a good approximation to be made when calculating the shadowing of the pixel. However this is only an approximation, and while it works for planar occluding objects and receivers it can lead to 'light bleeding' artefacts when the variance is high, causing areas that should be in shadow to be lit or partially lit. Myers [Myers 2007] discusses some implementation issues with this technique and its suitability for implementing on Direct3D 10 hardware. The main point being that the filtering precision on a Direct3D 10 implementation can be 32-bit instead of 16-bit. This is important because the computation in this algorithm is unstable and the precision has to be managed. The precision and numerical stability was also discussed in Donnelly and Lauritzen [Donnelly and Lauritzen 2006] but due to hardware constraints at the time they could only implement 32-bit precision by splitting depth values into two 16-bit values for storage and then recombining these afterwards.

Lauritzen [Lauritzen 2007] presents summed-area variance shadow maps (SAVSM), which build on VSM to reduce aliasing resulting in smooth soft shadow edges.

Lauritzen and McCool [Lauritzen and McCool 2008] present layered variance shadow maps (LVSM) which use multiple shadow map layers to remove the light leaking artefacts present in VSM. Each layer represents a reduced depth range so the resolution of each layer need not be so high as to maintain adequate rendering quality. LVSM can be used with 16-bit precision which is more readily available in graphics hardware.

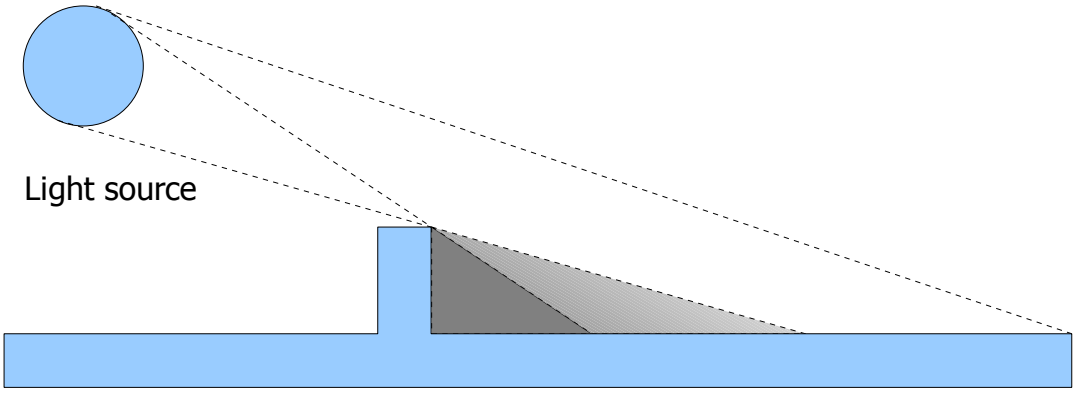

Figure 3.7.c: Penumbra illustration

Figure 3.7.c illustrates a penumbra, the partial shadow between a shadowed and lit area giving a soft edge to shadows at a distance. Notice that as the distance between the occluding and the receiving object increases, the penumbra will be larger and more gradual. The distance and the radius of the light source also contribute to the size and smoothness of the penumbra.

Fernando [Fernando 2005] presents percentage-closer soft shadows (PCSS), which model the penumbra so that it varies depending on this distance. This is an important addition to modelling shadows as it adds visual clues to the positioning of objects in a scene. Where objects are in contact with one another (such as a character standing on a flat floor casting a shadow onto it) the shadow edge will be hard (small penumbra) near the feet of the character. As the distance increases between the upper body of the character and the floor the shadow will get softer edges (large penumbra). Bavoil [Bavoil 2008] presents an implementation of the PCSS technique, which can be found in the NVidia Direct3D SDK [NVidia 2007] under 'Percentage Closer Soft Shadows'. Myers *et al.* [Myers et al. 2008] discusses the same algorithm with more detail regarding integration into a game engine. Lauritzen [Lauritzen 2007] discussed combining the PCSS technique with SAVSM to achieve a varied penumbra with reduced aliasing artefacts.

Annen *et al.* [Annen et al. 2007] introduce convolution shadow maps. Instead of using a binary test to decide if a pixel is in shadow (between the depth stored in the depth map and the distance from the pixel to the light) they attempt to apply a linear filter to the shadow map. Regular texture filtering applied to a shadow map does not result in filtering the end result of the shadow test, so instead they suggest a pre-filtering method.

This method involves converting the shadow map values by using a basis function. The resulting convolution shadow map can then be pre-filtered resulting in a filtered end result. They use a Fourier expansion to define the basis function. This approach allows for a kernel function to determine the shadowing of a pixel. This gives a smooth edge to shadows and does not suffer from aliasing artefacts like PCF. The main disadvantage with convolution shadow maps is that they are a computationally demanding technique and a real-time solution may not be achievable in some scenarios.

Inspired by these convolution shadow maps Annen *et al.* [Annen et al. 2008] developed exponential shadow maps (ESM). Like in convolution shadow maps, ESM uses a basis function to allow pre-filtering of the shadow map. Where this method varies is in the assumption that the distance to the light is greater than or equal to the distance stored in the shadow map. This assumption allows for an exponential expansion approximation which drastically reduces the computational complexity and allows this method to out perform convolution shadow maps. This assumption holds true in most cases, but not all, resulting in some artefacts. As a solution Annen *et al.* suggest a simple-to-calculate fall-back to an alternative technique such as PCF.

A combination of ESM and VSM was briefly discussed by Lauritzen and McCool [Lauritzen and McCool 2008]. The exponential variance shadow map (EVSM) produced 'promising results with good performance'.

Scherzer *et al.* [Scherzer et al. 2007] introduce a shadow mapping technique that makes use of old shadow map information from previous frames. They use the aptly named history buffer to store the previous shadow information which can then be used in the following frames. This approach is an efficient and computationally cheap method to produce accurate hard-edged shadows.

Shadow maps are usually used instead of shadow volumes for the large-scale real-time scenes often present in many games. This is because they are usually less computationally complicated to implement so real-time speeds are more easily achieved. There is also a substantial amount of work that focuses on improving shadow mapping techniques for large scenes.

Zhang *et al.* [Zhang et al. 2007] discuss parallel-split shadow maps (PSSM), which use multiple shadow maps in parallel spaced at different distances with each having a different resolution depending on the distance to the viewer. This means nearby shadows can be calculated from a high-resolution shadow map whereas distant shadows can be

calculated from a low-resolution shadow map. This level of detail approach saves valuable processing time in large scenes.

Dimitrov [Dimitrov 2007] describes cascaded shadow maps (CSM), which are similar to PSSM and build on them by taking into account view frustum culling. An implementation of the CSM can be found in the NVidia OpenGL SDK [NVidia 2007] under 'Cascaded Shadow Maps'.

Some of the shadow mapping techniques can be combined into hybrid methods and benefit from the advantages of each. In a recent release (June 2010) of the DirectX SDK [Microsoft 2010], CSM has been implemented with PCF (CSMPCF) and with variance shadow mapping (CSVM). These are available as samples in the SDK (under 'CascadedShadowMaps11').

As we have discussed there are a variety of advanced shadow mapping algorithms which provide effective solutions to shadowing. Some of these techniques can be used in combination to create hybrid algorithms which benefit from multiple technique's advantages, resulting in improved solutions.

3.8 Smoothies and Penumbra Wedges

Many of the advanced shadow volume and shadow mapping algorithms for creating smooth soft edged shadows are computationally expensive. The following methods offer alternatives and approximations for creating smooth edged shadows which may be able to offer improved performance for plausible soft shadows.

Penumbra wedges [Akenine-Moller and Assarsson 2002] offer an approximation to add soft edged shadows to the initial shadow volume technique. The basic idea is to extrude a penumbra wedge in place of where the silhouette edge usually gets extruded. This is done by extruding the silhouette edge twice, in two directions off a tangent to the light source which is of course dependent on the radius of the light source, to form the wedge shape. Once the wedge is created a pixel inside the wedge can calculate the approximate shadowing depending on its position inside the wedge. Given modern hardware, this method should now be capable of real-time speeds.

Chan and Durand [Chan and Durand 2003] present smoothies, a technique that builds on the shadow mapping algorithm to remove the aliasing artefacts at the edges of the shadow. Unlike the other soft shadow map algorithms, that usually perform different operations to the shadow map or sample from it differently, smoothies hide the rough

shadow edge by building a smooth fake edge. Although this method could be considered an inaccurate representation of shadows it does provide a smooth edge to shadows with real-time performance. The model works by building a standard shadow map, storing the depth values of the occluding objects in the scene. Then the silhouette edges of the objects are extruded and filled to create the smoothies. The smoothies are rendered to separate buffers storing a depth map and an alpha map of the values. Finally the final image is created using the information from the maps to create smooth edged shadows. This method builds smoothies from the silhouette edges so it might be possible to implement this technique on the GPU using the geometry shader in a similar fashion to the way it was used in the 'ShadowVolume10' sample found in the DirectX SDK [Microsoft 2010] (and SDK versions from later than December 2005).

Wyman and Hansen [Wyman and Hansen 2003] introduce penumbra maps which map out the approximate penumbra regions by calculating the silhouette edges of shadow casting objects from the light source. First a shadow map is created storing the depths of objects to the light source. Then a penumbra map is generated which stores the shadow intensity at the silhouette edges. Finally the shadow can be rendered by combining the depth information from the shadow map and the shadow intensity from the penumbra map. This method can produce soft edged shadows with a varying size penumbra similar to that achieved by PCSS.

Chapter 4 Making Smoke

Opaque solid models and effects can be used to build many scenes and virtual worlds in computer graphics but cannot be used convincingly to simulate and render transparent volumes. Another approach is required when we want to render smoke or similar transparent non-solids. In Chapter 4 we look at how smoke can be simulated and rendered in computer graphics and then we discuss the addition of shadows.

4.1 Simulation

The main techniques for the simulation of smoke can be broken down into different categories and include particle-based Lagrangian systems, grid-based Eulerian methods, and hybrids.

The accuracy required for the simulation of smoke depends on the specific application it is being designed for. Some approaches derive the simulation directly from the Navier-Stokes equations, Stam [Stam 1999]. Other approaches, Fedkiw *et al.* [Fedkiw et al. 2001], use the inviscid Euler equations, which provide a good model for smoke because smoke has a low viscosity, which can be considered negligible. For the application of computer games where real-time speeds are critical, simple and efficient methods are preferable.

4.1.1 Grid-Based Eulerian Approach

A grid-based Eulerian approach to smoke simulation involves modelling a theoretical box volume, which the smoke will occupy. The volume is divided into voxels (volumetric pixel – small equally sized cubes) with each voxel containing information about the density, pressure, and other factors that are used to build up a simulation for a smoke model. In a simulation step the content of a voxel is calculated from nearby neighbouring voxels or by an advection technique as in the ground-breaking work on Stable Fluids by Stam [Stam 1999] [Stam 2003]. Stam discusses a stable grid-based model for fluid simulation which has been a basis for many other models in this area since. This proposed model suffers from numerical dissipation, which causes the flow to dampen too quickly, nevertheless the proposed model still provides a suitable technique for graphics applications such as games where accuracy is not paramount.

Fedkiw *et al.* [Fedkiw et al. 2001] introduce a vorticity confinement technique to improve the accuracy of the model proposed by Stam. Vorticity confinement adds back the lost energy due to numerical dissipation and this technique has been noted for producing 'some of the most visually complex fluid simulations to date' [Molemaker et al. 2008].

These vorticity confinement methods can produce some very realistic simulations of smoke, however, as usual, there is a trade-off between the accuracy of the simulation and the computational workload. Still many of these methods can be run in real-time on modern hardware.

Molemaker *et al.* [Molemaker et al. 2008] propose the use of the QUICK (Quadratic Upstream Interpolation for Convective Kinematics) advection scheme to minimise the numerical dissipation. This method preserves the small-scale turbulent flows without needing to implement vorticity confinement techniques, which 'cannot fully compensate for excess numerical dissipation' [Molemaker et al. 2008].

Crane *et al.* [Crane et al. 2007] discuss a grid-based smoke simulation method that physically responds to movement of 3D objects. An implementation of their method can be found in the NVidia Direct3D SDK [NVidia 2007] under 'Smoke'. They make use of the MacCormack advection scheme discussed by Selle *et al.* [Selle et al. 2008].

A disadvantage with all the grid-based methods is that the smoke volume is confined to a limited size grid that requires some sort of boundary conditions to confine the smoke volume. This has been overcome in recent research [Cohen et al. 2010] by use of a hybrid method combining a grid-based approach with a particle-based approach which we discuss below (see section 4.1.4).

4.1.2 Particle System-Based Lagrangian Approach

The particle system Lagrangian approach involves modelling smoke as a group of particles, the concept introduced by Reeves [Reeves 1983]. Particles are born in an initial position and state and then move according to a set of rules or equations until they are destroyed or are reset. Particle systems can be used to model a wide range of fuzzy objects and are flexible by adapting the rules that govern their simulation. The constraints of this approach include the quantity of particles that can be processed in real-time and the complexity of the rules for the simulation.

The nature of particle systems means that they are well suited to the SIMD (single instruction multiple data) architecture of GPUs. A single instruction can be used to govern the many particles in a particle system. The parallel processing capabilities of modern GPUs provide a vast improvement to simulation speeds for most particle systems. For smoke simulation via a particle system a GPU implementation is advisable to reap the benefits of parallel processing.

A common method for a GPU implementation involves using textures with particle data stored in the colour channels in each texel (texture element). This approach was used by Latta [Latta 2004] to create a GPU-based particle system containing roughly one million particles. This was achieved by using texture sizes of 1024 by 1024 to store the particle data while the particle simulation was performed inside the pixel shader. Kolb e*t al.* [Kolb et al. 2004] built on this by adding support for collision detection with geometry.

Now with the geometry shader and the stream out function in Direct3D 10 [Blythe 2006], particles can be stored as vertices with simulation operations performed in the geometry shader and streamed out. Also because the geometry shader can add or remove vertices from the stream, particles can be created or destroyed in the geometry shader. An implementation that uses this approach can be found in the DirectX SDK [Microsoft 2010] (and SDK versions from later than December 2005) under 'ParticlesGS'. The sample demonstrates how a particle system can be entirely encapsulated on the GPU in Direct3D 10. The simulation is processed in the geometry shader with a stream of vertices representing each particle. The geometry shader can create or destroy particles stored as vertices and the stream can be passed out before going any further down the pipeline. On the rendering pass particles in the stream are expanded to form billboards by the geometry shader and then they are additively blended to the scene.

Latta [Latta 2007] presents a useful overview on particle systems, briefly describing the main ways to implement them on the CPU and GPU and key considerations when developing them for use in games.

4.1.3 Smoothed Particle Hydrodynamics

Smoothed-particle Hydrodynamics (SPH) is a Lagrangian method of simulating fluids. The system is built from a set of complex particles usually storing information in attributes for use in simulation. Particles have a spatial distance over which their attribute values are smoothed. A kernel function is used to determine the distance and the amount

of smoothing that takes place. This means the attribute values of a particle in a particular position can be determined by looking at the nearby particles as denoted by the kernel function. On a side note, there are similarities between SPH and the Boids [Reynolds 1987] flocking model. The kernel function introduces local relationships between particles, similar to the local relationships in Boids (separation is similar to pressure, and alignment is similar to viscosity). Particles that are further than a set distance away (from the measured attribute 'position'), determined by the kernel function, contribute nothing to the measured property so that they can be ignored and calculations do not scale at an exponential complexity rate. The kernel function can change depending on factors such as particle density, which can also reduce complexity by making the kernel ignore particles outside a smaller nearby area for high-density areas (therefore reducing the number of lookups).

The work by Muller *et al.* [Muller et al. 2003] on fluid simulation is based on SPH, which they describe as ‘an interpolation method for particle systems’. They suggest using a particle system because it simplifies the Navier-Stokes equations. They write “In contrast to Eulerian grid-based approaches, the particle-based approach makes mass conservation equations and convection terms dispensable which reduces the complexity of the simulation”. Reducing the complexity of the simulation means that there is less to calculate at run-time so the model can run quicker, which is a boon for real-time applications.

4.1.4 Hybrid Methods

Cohen *et al.* [Cohen et al. 2010] use a hybrid approach to achieve a complex and detailed model for smoke and dust trailing from the back of a virtual model car. A grid-based approach is used for dust in close proximity to the back of the virtual car. A box volume is used to contain the grid-based simulation, which moves relative to the car (similar to the way a bounding box encapsulates a model for collision detection algorithms). Any dust that reaches the edge of the volume is transferred into a particle system allowing the new particle dust to move freely outside of the box volume. This hybrid method takes the accuracy and fine detail of a grid-based approach and combines it with the boundless freedom of a particle system. For the grid-based part of the simulation the MacCormack advection scheme discussed by Selle *et al.* [Selle et al. 2008] is used.

4.2 Rendering

4.2.1 Volume Rendering

Volume rendering is not a major topic for discussion in this thesis due to the early choice of simulating smoke via particle systems. It does however offer an alternative solution for simulation and there is plenty of ongoing research in this field which the reader may find interesting.

Kruger and Westermann [Kruger and Westermann 2003] present a volume ray-casting technique for rendering volumes using DirectX 9 level graphics. They utilise the SIMD architecture of the GPU via textures and the pixel shader to accelerate volume rendering. The volume ray-casting technique involves casting rays (one per pixel) from the camera viewpoint through the volume clipping it to the edges of the volume. Then a number of sample points are taken at intervals along the cast ray (through the volume). The samples are shaded and then amalgamated to arrive at a final value for that individual pixel.

Ikits *et al.* [Ikits et al. 2004] present some typical volume rendering implementations and some advanced techniques for volumetric lighting, shadows, and some light scattering through translucent media.

4.2.2 Particle System Rendering

We have discussed the methods for simulation of a particle system, some more involved than others, but either way the particles need to be rendered to make the system visible and achieve the required effect. Each particle in the system can be rendered as a single pixel or an object. The most common method in games is to render the particle as a billboard. 3D meshes or metaballs can also be rendered in the place of particles.

Selle *et al.* [Selle et al. 2004] use a billboard approach to achieve a cartoon style output rendering of smoke from their particle system. In addition to the standard billboarding technique they also use a technique for edge detection so the edges of the render can be drawn in black adding to the overall effect of the carton style rendering. These sharp silhouettes are also drawn in where there is a big change in the depth between particles so as to highlight interesting points in the smoke volume.

4.2.3 Particle Blending and Sorting

When we render particles to the scene there are a number of blending options available. Additive and alpha additive blending add the particle colour to the colour already present in the scene (as long as the depth test is passed). Particle colours are added to the scene irrelevant of their individual depths and order which has no effect on the final result. Where particles overlap one another additive blending can lead to undesired results as the resulting colour is an equal weighting of every overlapping colour, which is unrealistic for translucent media. To get a more accurate result alpha blending can be used but the requirement of alpha blending is that the particles must be sorted in the correct depth order so that particles in the distance are rendered first, allowing the front particles to be blended correctly. Unfortunately sorting on the GPU is not trivial and it is computationally demanding, however, there are methods for GPU sorting that can run in real-time and various optimisations that can improve performance. Kipfer and Westermann [Kipfer and Westermann 2005] discuss implementing sorting algorithms on the GPU. They examine a simple implementation of the *odd-even transition sort*, which operates by comparing the data in the even positions with data in the odd positions and switching them if they are not in order, then comparing odd data with the even data, and repeating until the data set is sorted. Odd-even transition sorting can work for small data sets (small number of particles) but when we want to deal with large particle systems we need a faster algorithm or approximation to keep the simulation running at real-time speeds. Kipfer and Westermann [Kipfer and Westermann 2005] go on to discuss the *odd-even merge sort*, which is a faster algorithm, taking 210 passes to sort a data set of 1 million elements (or 1 million particles), whereas the *odd-even transition sort* takes 1 million passes for the same data set. Although 210 passes would likely take too much computation time per frame to be performing real-time, the algorithm can be split down to perform some of the passes each frame. As a result we do not have a perfectly sorted set of data every frame but the data set becomes closer to being sorted every frame and the performance is real-time, which is often an acceptable solution for particle systems. Kipfer and Westermann [Kipfer and Westermann 2005] then describe a more efficient GPU implementation using a *bitonic merge sort*. This algorithm has an advantage of performing a complete sort faster than the *odd-even merge sort* but a disadvantage of not approaching a complete sort in a smooth way and hence is not suitable for splitting the computation over multiple frames. To keep the simulation running in real-time the size

of the data set must be kept small enough to be completely sorted within the time one frame is rendered. Sorting smaller sets of data tends to be much faster, so many performance enhancing methods involve splitting large data sets down into multiple smaller sets wherever possible.

Another technique to improve the performance of sorting for particle systems is the half-angle axis sorting approach discussed by both Ikits *et al.* [Ikits et al. 2004], and later by Green [Green 2008]. Green requires a set of sorted particles in both the light direction and the camera direction so that correct self-shadowing and correct blending can be performed. The method that he proposes is to sort the particles along the half-angle between the camera and the light, tackling two expensive sorting requirements with only one sorting operation. Another technique he uses is to split the particles into batches arranged as slices perpendicular to the half-angle axis so that the particles can be rendered in smaller batches. Slices can also be used to split up the size of a data set before sorting individual batches. Splitting up a data set in this fashion is often called a *bucket sort* whereby data is split into different 'buckets' (subsets). This divide and conquer approach is efficient for breaking down large data sets to get a roughly sorted data. If a complete sort is required a different algorithm such as the *bitonic merge sort* could be used on each bucket. Hybrid methods like this or the similar hybrid method employed by Sintorn and Assarsson [Sintorn and Assarsson 2007] are often very efficient and there are a variety of other sorting algorithms that may be advantageous in a hybrid approach. With a set of sorted data we can perform alpha blending and render particles in a back to front order so we achieve correct blending. With unsorted data we can use additive blending or alpha additive blending instead.

4.2.4 Overdraw in Particle System Rendering

Overdraw is a problem which occurs when a screen pixel has to be re-rendered multiple times due to multiple translucent objects on top of one another. Overdraw often plagues particle systems that are rendered with large translucent billboards. Each particle requires rendering and if these particles are large and numerous the amount of pixels which require shading is likely vast. Cantley [Cantlay 2007] discusses some techniques to tackle overdraw in particle systems by introducing a mixed resolution technique. Another simpler solution is discussed by The Valve Developer Community [The Valve Developer Community 2010] whereby particle billboards are blended out and then not rendered depending on how much screen space they take up. This is a similar idea to the soft

particle technique, however, with soft particles each pixel is blended out where it is positioned closely to 3D geometry. The soft particles technique is discussed in detail in the following section.

4.2.5 Soft Particles

When a particle is rendered as a 2D billboard and intersects 3D geometry a common artefact occurs; a hard edge is seen at the intersection point. This can be rectified by a technique called soft particles. The idea behind soft particles is to make the particle more transparent as it gets closer to intersecting with 3D geometry. Recording the depth information from the 3D geometry in the scene and then comparing to the depth of the particle is a commonly used technique to solve this.

Lorach [Lorach 2007] discusses the soft particles technique in a GPU implementation. First the depth values are needed, and Lorach suggests two possible solutions; depth information can be stored on a first pass using the depth buffer, or alternately the depth information can be stored in a separate render target whilst rendering background objects in the scene (objects behind the particle system effect). Using the MRTs suggestion is likely to be a more efficient and flexible solution, because the stored depth information could be reused in other effects. The second pass deals with the rendering of the particle system. The geometry shader is used to expand a set of particle positions into two triangles forming the canvas for the billboard. In the pixel shader the depth values can be compared and the particle can be made more transparent as it approaches any of the stored depths. For this Lorach suggests using a custom contrast function to fade instead of a linear fade between the particle's maximum opacity and completely transparent.

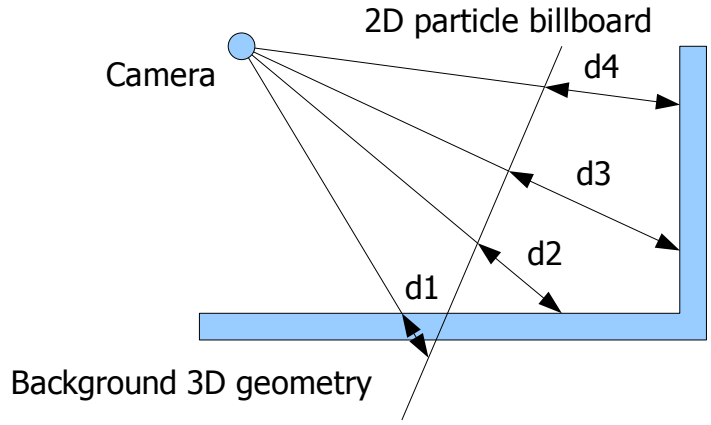

Figure 4.2.5.a: The soft particles technique

Figure 4.2.5.a the numbering shows the difference of the depths, which are used to calculate the opacity of the pixels in the 2D particle. Difference d1 is negative so the pixel is ignored, d2 and d4 are small so the pixels here are slightly faded, d3 is large so the pixels is at the particle's maximum opacity.

The 'SoftParticles' sample in the DirectX SDK [Microsoft 2010] (and SDK versions from later than June 2006) shows an implementation of the soft particles technique in action. It implements the basic soft particles model but also implements, soft depth sprites, and soft volumetric particles. Depth sprites are simply particles that have additional depth information. This depth can be used in the soft particles technique when calculating the opacity of the particle.

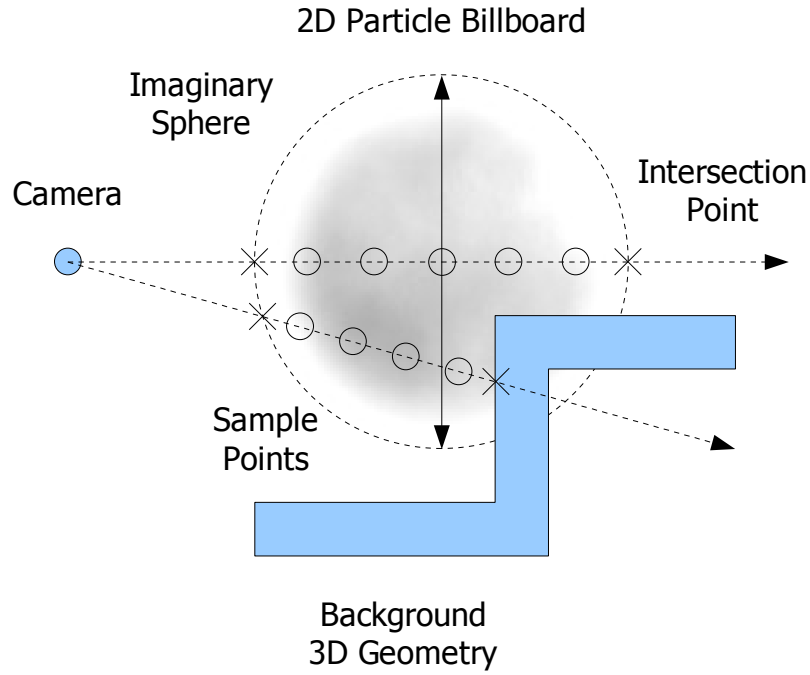

Figure 4.2.5.b: The soft volumetric particles technique

Volumetric particles offer a more complicated technique for rendering which does not suffer from the artefacts caused by 2D billboards intersecting with 3D geometry in the same way. However, the soft particles technique can still be implemented on volumetric particles. Particle billboards are encapsulated in an imaginary sphere. When rendering a pixel on the billboard, a ray is cast from the camera through the pixel. Where this ray intersects the sphere in two places a set of sample points are taken through the sphere in-between the intersection points with each point mapping to a 3D noise texture, based on the concept of hypertextures [Perlin and Hoffert 1989]. A ray march through the sample points calculates the opacity and the lighting for the pixel. The addition with the soft volumetric particles implementation adjusts to take only samples between the first intersection point and the depth buffer if the depth buffer intersects the ray before the second intersection point on the sphere.

4.2.6 SPH Rendering

Smoothed particle hydrodynamics systems are often rendered in a different way to the standard particle systems. SPH systems can be converted to a 3D grid and then a volume-rendering technique can be used. This is often an expensive method, which can cause problems with keeping the simulation and rendering real-time.

The previously discussed work by Muller *et al.* [Muller et al. 2003] covers both splatting and marching cubes rendering methods for SPH. They test both for performance and conclude that splatting runs faster than the marching cubes but the marching cubes method produces a more realistic output.

Fraedrich *et al.* [Fraedrich et al. 2010] present a volume rendering technique for SPH. Particles are re-sampled to a 3D grid in view space, which they call the 'perspective grid'. A kernel function is used to re-sample the SPH data to the perspective grid and then a GPU-based volume ray-casting technique is used to render the re-sampled data.

Chapter 5 Homemade Smoke and Shadows

This chapter discusses our main contribution to research; casting shadows onto a volume of smoke particles.

5.1 Particle System Development

Our implementation uses a DirectX 10 GPU-based particle system with particles stored as vertices and simulation through the geometry shader and stream out sections of the pipeline. The particle simulation employs a similar approach to the implementation found in the 'ParticlesGS' sample found in the DirectX SDK [Microsoft 2010] (and SDK versions from later than December 2005).

The nature of the geometry shader in the Direct3D 10 pipeline means that new vertices (particles) can be created on the fly, so a range of new particle system effects can be created. ParticlesGS shows an example of how exploding fireworks can be created with this new capability.

Our particle system runs entirely on the GPU and particles can be created and destroyed in the geometry shader, so there is no need for CPU intervention. Keeping the particle system on the GPU allows for faster speeds and frees the CPU up for other tasks.

There are two common GPU particle system implementations; one that stores particles as texels in a texture and updates through a pixel shader, and another that stores particles as vertices in a vertex buffer using the geometry shader to perform updates and stream out to make an early exit from the pipeline.

Our particle system stores particles as vertices using vertex buffers for storage. In particular, we use three separate vertex buffers for storage. One initial seeding buffer is used to start the simulation, holding an initial state. The other two are used as an input and an output buffer, which are switched every time particles are updated so that the output from the previous update becomes the input to the next update. Our initial seeding buffer is initialised to contain a single 'root' type particle, which is used as the spawn location for other particles in the system.

As mentioned above, particle information is stored in vertices, but the format in which this is stored is flexible. For example, most particle system simulations require particle positional information to be stored, so this could be done using a 3D vector with 32-bit

precision for each of the X, Y, and Z coordinates. A solution for this is to use a three component, 96-bit, floating-point format for the first element of the vertex input layout description allowing us to store position information for particles. If we want to store additional information such as the type, the colour, or the timer on a particle before it is reset then we have to add elements to this input vertex layout to allow for the storage of this information. There is a limit to how much data can be stored and therefore a maximum number of particles before the memory is exhausted, so careful design is important.

The parallel nature of a GPU vertex based particle system that creates and destroys a varying amount of particles per update, means that there is no way of knowing how many particles are in a scene at any one time. To handle draw calls a 'DrawAuto' feature is included in Direct3D 10 that can handle the varying amount of data written to the stream out vertex buffer to be used as the input data for the next draw call.

Particle systems can be designed to represent a vast variety of different objects, making them a very flexible tool in computer graphics for applications such as computer games. Particle movement is usually the most important part of the simulation, and particles can be made to follow curved paths made from splines, causing them to move away from or towards a position, or to accelerate or decelerate towards a position or in a direction. The possibilities are almost endless, the only constraints being the amount of available memory for particle data storage and the computational power of the GPU (if a real-time solution is important). Only storing the position of a particle limits the complexity of movement possible. By storing additional positional data such as a velocity vector, an acceleration vector, or even multiple positions that form a spline and that are accompanied by a timer, particles can be made to move in much more complex patterns.

An important component in the movement of particles is the element of randomness. Without some randomness in the movement or some randomness when a new particle is spawned, all the particles will be moving in the same way and the system as a whole would look too simple with all the particles following each other. Adding some randomness gives the illusion that the system is more complex than it actually is and adds some differences to each particle. Getting different random numbers for each particle in a system is not a trivial task on the GPU. A pseudo-random number generator (often abbreviated as PRNG) can be used for some tasks, taking an input seed value and returning a pseudo-random number resulting from some equation. However, this

approach can result in some unwanted patterns when used in particle systems for movement. Howes and Thomas [Howes and Thomas 2007] discuss and compare some different methods for PRNG on the GPU using CUDA [NVidia CUDA 2007].

Another approach to getting random numbers onto the GPU is to fill all the texels in a texture with random numbers using the CPU and then pass the texture as a resource to the GPU. The random numbers in the texture can then be sampled and used. The texture is filled with new random numbers every frame from the CPU. This method can be used to provide a consistent flow of good random numbers.

Figure 5.1.a: Particle system with particles following splines to form a sphere

Figure 5.1.a shows particles following randomly generated splines forming a spherical shape. Particles spawn randomly on a large radius sphere and then move towards the centre following a spline.

Figure 5.1.b: Particle systems forming smoke plume shapes

Figure 5.1.b shows particles following randomly generated splines to form smoke plume shapes.

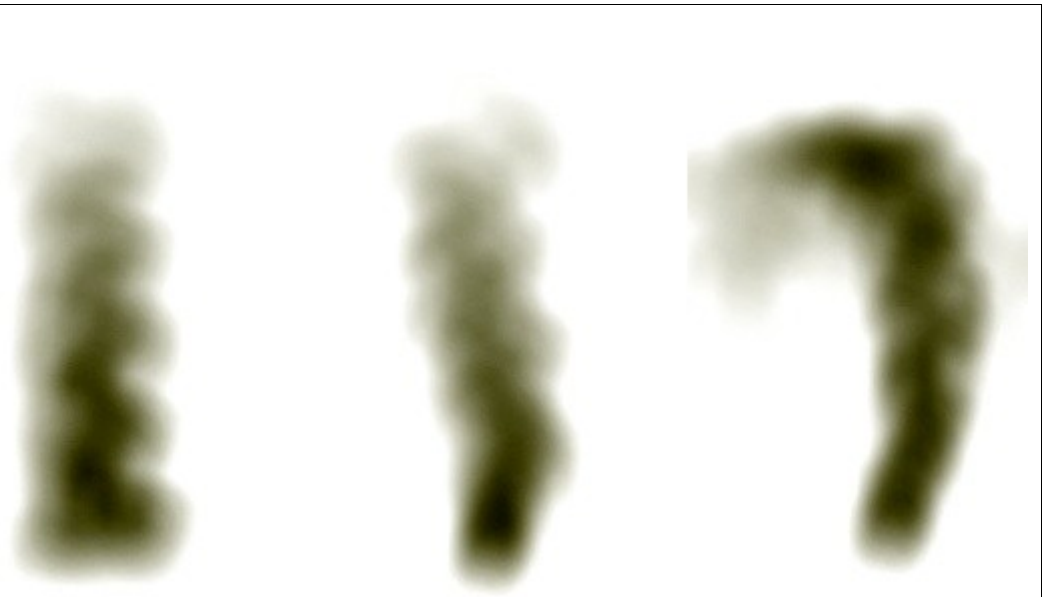

Figure 5.1.c: Particle systems forming smoke plume shapes with texture

Figure 5.1.c shows particles following randomly generated splines to form smoke plume shapes in which particles are rendered with a large radius and with a smoke texture.

For our purposes a complex particle system simulation is not a priority; in fact, a very simple simulation with particles just drifting in a direction with some random vectors can

be enough to represent the buoyancy of hot smoke drifting out of a vent, or the changes in wind direction. Whatever the scene, smoke simulation in real-time applications (such as games), is usually kept simple and efficient. Our work uses a simple particle system to simulate the smoke, in which particles are spawned in a grid from a ground plane and drift upwards along the y-axis before being removed from simulation. Although this is not a physically accurate model for a smoke volume, it allows for testing rendering techniques and shadowing, for which we present a novel idea for rendering shadows through the smoke particle system. A more complicated and physically accurate particle system could be used with our rendering technique with ease.

In our implementation we use alpha additive blending, therefore we do not need to sort the particles beforehand. Our shadowing method, which is discussed in the next section, can be used with alpha blending but this will require an additional sorting solution to achieve correct blending. The *odd-even merge sort* with sorting passes split over multiple frames would be a suitable solution and has been used successfully in related work by van Pelt *et al.* [van Pelt et al. 2010] and direct to video [direct to video 2009].

5.2 Adding Shadows

Shadowing of a particle system adds a visual element to its position and shape, helping to make it look natural and integrate into the scene rather than looking out-of-place. There are different kinds of shadowing resulting from different shadowing techniques. We will refer to them using the following terms:

Cast shadows – are shadows that are cast by the particle system and are visible on other solid objects in the scene.

Self-shadows (or self-shadowing) – are shadows that are cast by the particles that appear on other particles in the same particle system.

External shadows – are shadows that are cast by other solid models in the scene and land on the particle system.

The use of shadow maps is a common approach to implementing cast shadows from particle systems. Unlike casting shadows from solid opaque objects, particles are often represented as translucent billboards therefore requiring shadowing techniques to be implemented differently and the translucency becomes an issue. These difficulties can make shadows look unrealistic and therefore a more advanced shadow approach may be required. Cast shadows often tie in with the self-shadowing of a particle system where the resulting self-shadowing information, often stored as a shadow map, can be reused to create cast shadows.

Most existing methods involving self-shadowing of particle systems require particles to be sorted along an axis so that the opacity and shadowing information can be accumulated for each particle in order. This process is computationally demanding but there are optimisations, which can reduce the complexity. Green [Green 2008] performs real-time self-shadowing of a particle system and discusses some techniques for reducing the complexity, including the half-angle slice rendering technique, which was discussed in the previous chapter.

Deep shadow maps [Lokovic and Veach 2000] can be used to add self-shadowing to particle systems where particles are translucent. Where as regular shadow maps store a single depth value for each pixel, a deep shadow map stores a ‘visibility function’ which estimates the amount of light that passes through at different depths. Unfortunately deep shadow maps are computationally expensive and cannot guarantee a real-time solution.

Opacity shadow maps [Kim and Neumann 2001] provide a real-time solution to self-shadowing. Opacity maps are spaced out throughout the volume and face towards the light source; cutting the volume into slices. Each is rendered from the point of view of the light source summing and storing the alpha values of the particles contained up to the depth of the opacity map (any particles outside the depth of the slice are clipped). These opacity maps store opacity values at different depths through the volume which can be used to calculate the shadowing at different depths when rendering the volume. While this technique does provide a real-time solution it does introduce artefacts due to linear interpolation between opacity maps. The layering artefacts can be reduced with a greater number of opacity maps but obviously at a cost of speed.

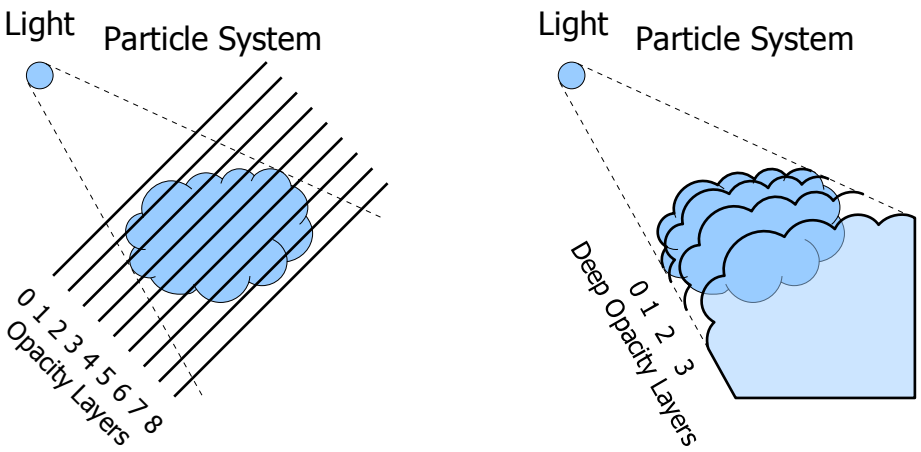

Figure 5.2.a: Opacity shadow maps and deep opacity shadow maps

Figure 5.2.a shows the similarities between opacity shadow maps (left) and deep opacity shadow maps (right). Deep opacity maps [Yuksel and Keyser 2008] offer a solution which does not suffer from layering artefacts and yet runs at real-time speeds. First a depth map is created from the point of view of the light source, which captures the depth of particles nearest the light source. Secondly, opacity maps are created, storing the opacity at different depths through the volume, but instead of using flat parallel slices for each layer, the depth map dictates the first layer and further layers are at an increment to this depth map. The difference between the layers is illustrated in Figure 5.2.a. This adjustment improves the accuracy of the shadowing and does not need as many layers as the opacity shadow maps technique to achieve an improved quality final image. With these recent techniques, real-time self-shadowing of particle systems has become feasible

for mainstream consumer graphics cards, and they open up some interesting future possibilities for hybrid techniques.

To our knowledge, external shadowing of particle systems has not been the focus of any other research, so shadowing information from solid objects currently in a scene has no effect on a particle system. A particle system which represents smoke should interact with external shadows from other objects in the scene so as to look realistic and properly integrated into the scene. Figure 5.2.b shows our GPU vertex based particle system without any form of shadowing. Particles are expanded in the geometry shader and rendered as point sprites with alpha additive blending which does not require any sorting. An external shadow is cast from the dwarf model, which should be seen cast onto the smoke particle system as well as on the wall.

Figure 5.2.b: GPU particle system without external shadowing effect

We have experimented with both shadow mapping and shadow volume techniques for implementing external shadows. We found shadow mapping to be a more suitable solution in this case as it is a simple operation to determine if a particle is in shadow or not and we can easily pass the shadow map as a texture input when rendering our GPU

vertex based particle system. We sample and compare the depth in the shadow map to the actual depth of the particle in the pixel shader to determine if the particle pixel is in shadow and render the correct colour accordingly. Figure 5.2.c shows our results using this technique. The test application runs at real-time speeds on modern consumer graphics hardware, Figure 5.2.c was rendered at 145 fps on a NVidia GeForce 8800 GTS graphics card. The light source in Figure 5.2.c is modelled as a single dynamic point light which followed a circular path in this test scene.

Figure 5.2.c: GPU particle system with external shadowing effect

Figure 5.2.d: Different view showing our external shadow implementation

Figure 5.2.d shows our model from a different camera angle. We presented this method as a poster at the Eurographics 2010 conference [Bass and Anderson 2010]. A copy of the poster can be found in Appendix A.

Figure 5.2.e shows a detailed flowchart describing the processes that take place in rendering a single frame using our technique. For the particle system simulation we use a GPU vertex based particle system in which a seeding buffer is used to initialise the system. Input and output buffers are used afterwards to store the particle system simulation data which are swapped each frame. Once the particle system has been updated, the remaining scene objects are updated. A shadow map is created and the scene objects are rendered with shadows. The shadow map is reused when rendering the particle system to apply external shadowing to the particle system.

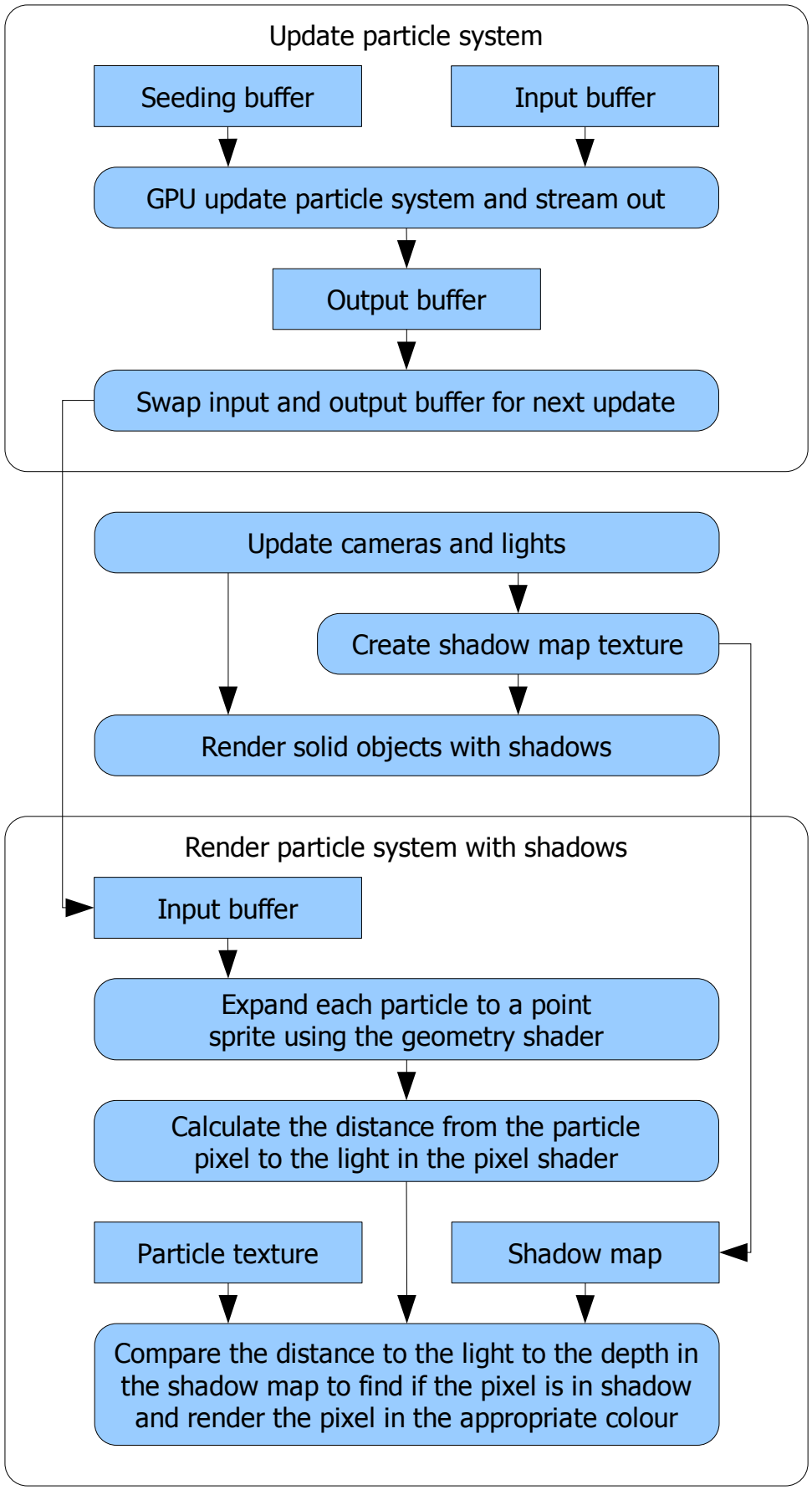

Figure 5.2.e: Flowchart of the processes in rendering a frame of our model

5.3 Improving our Method by Incorporating Existing Techniques

The current implementation shows a lot of room for improvements. This section will discuss some ideas for future research and development.

The basic shadow map algorithm suffers from aliasing artefacts resulting in rough pixelated edges to shadows. By using shadow mapping our method also suffers from these artefacts throughout the 3D shadow. However, as we have discussed in a previous chapter, there are several available advanced shadow mapping techniques that successfully remove such artefacts and result in a smooth edged shadow. Combining one of the more advanced shadow mapping techniques with our external shadowing should be a trivial task. The PCSS [Fernando 2005] technique would be a particularly interesting addition as it has a variable size penumbra which would be emphasised by our 3D shadow effect seen through the particle system. Sharper edged shadows would become more blurred for particles further away from the light source, adding to the realism of the scene.

Soft particles is a technique to remove hard edges where a 2D billboard intersects 3D geometry. In our current external shadowing technique there are hard edges where the shadow cuts through particles. This is partly caused by the inaccuracies in the shadow mapping technique and the sharp drop off which we can also address with an alternative, more advanced, shadow mapping algorithm (like PCSS [Fernando 2005]).

The soft particles technique uses the scene depth buffer when rendering particles and as the distance between the particle and the depth buffer shortens, the particle is alpha blended out. We can use the soft particles technique in our model to alpha blend particle pixels which intersect 3D geometry as normal but we cannot use the soft particles technique directly to help remove any artefacts caused by 2D particles intersecting our shadow map. However we can implement a similar technique which uses the depths stored in the shadow map to blend particle pixels.

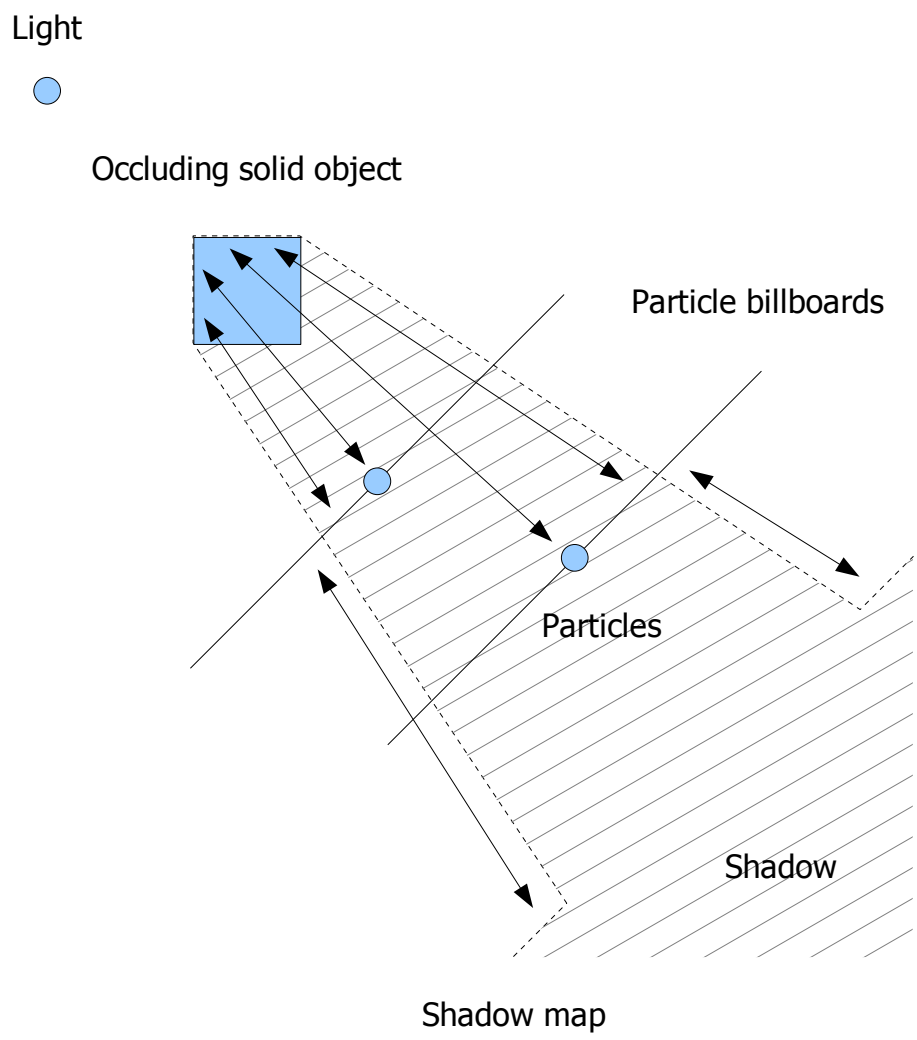

Figure 5.3.a: Soft particles for our technique

Figure 5.3.a shows how we have implemented a soft shadow technique using a shadow map. Notice that the particle billboards in Figure 5.3.a are partially in shadow and partially lit. In our original shadowing a hard edge appears at the cut-off between pixels in shadow and lit areas. In the vertex shader we have sampled the shadow map, calculated the distance between the particle position and the shadow map sample and passed the result to the pixel shader. In the pixel shader we repeat a similar process to calculate the distance between the pixel and the shadow map sample. With this data we can calculate the pixel shading using a simple weighting function between the two depths, so as the depths vary the pixel can be smoothly shaded from being in full shadow to being fully lit. The shadow map tends to vary fairly quickly and consequently the shading will also, but to counter this the shadow map could be blurred beforehand.

Figure 5.3.b: Our soft particles

Figure 5.3.b shows the result we achieve from using this technique with a shadow map. We could use this technique of comparing the per vertex depth and the per pixel depth with a more complex shadow mapping model, but we will leave this for future research. Another possibility for reducing the hard edge artefacts is to look at using volumetric particles to represent smoke. Figure 5.3.b was rendered at 124 fps on an ATI Radeon HD 5770 graphics card.

Green [Green 2008] published a particularly relevant paper, describing a method for volumetric particle shadows, which includes a discussion for sorting on the GPU and a method for the self-shadowing of particle systems. Self-shadowing of particle systems

usually requires particles to be sorted along an axis. Green suggests using half-angle slice rendering [Ikits et al. 2004] and half-angle axis sorting for the basis of his method. By sorting particles along the half-angle axis (between the light and the viewer), all the information for accumulating shadows and blending is available, saving on the computational complexity and the memory requirements. Another option for self-shadowing is a deferred shadowing approach like the method discussed by direct to video [direct to video 2009]. The deep opacity maps [Yuksel and Keyser 2008] method is another approach to self-shadowing which could be applied to particle systems, which looks to be a particularly effective approach. Deep opacity maps could be used instead of the regular slicing method discussed by Green [Green 2008] to build an improved solution.

Chapter 6 Final Conclusions

6.1 Summary of Contributions

We have presented an overview on lighting techniques, methods for smoke simulation and rendering with various shadowing techniques. We have added to the body of knowledge in this area by presenting two novel techniques.

6.1.1 External Shadowing of Particle Systems

Our main contribution is our method for casting external shadows onto particle systems. Our method can be used to achieve real-time 3D shadowing seen visually through a translucent particle system. We have successfully implemented this technique on a GPU vertex based particle system simulation using a typical billboarding approach for rendering particles to represent smoke. We have also discussed a modified implementation of the soft particles technique (inspired by Lorach [Lorach 2007] and the sample in the DirectX SDK [Microsoft 2010]) with the purpose of reducing the hard edge artefacts.

6.1.2 Surface-Curvature Ambient Occlusion

Our second contribution is Surface-Curvature Ambient Occlusion (SCAO); a method for estimating ambient lighting for opaque solid models. SCAO uses the geometry shader to provide a more accurate estimate of the ambient lighting term where often a constant ambient lighting term is used. It can be easily implemented into a lighting model with diffuse and specular terms providing a complete lighting model for 3D opaque solid models.

6.2 Discussion

The primary aim of our research has been to develop a real-time method that improves upon the realism of smoke rendering under different lighting conditions. Through a detailed exploration of the subject area we narrowed our focus to the casting of external shadows onto a smoke particle system.

Our resulting method provides an effective way to add external shadowing to a smoke particle system with aesthetically pleasing results. Our method opens up many interesting opportunities for further work.

During our research we developed another novel method, SCAO, using the geometry shader to calculate an estimate for ambient lighting reaching a surface.

6.3 Future Work

Our two novel techniques which we have introduced provide some scope for further development and future research.

6.3.1 External Shadowing of Particle Systems

We have already looked at using the soft particles technique to reduce the hard edge artefacts and while our modified soft particles implementation yielded some promising results, it did not completely resolve the artefacts. Perhaps a better solution might be to use a more advanced shadow mapping technique to determine the shadow of a particle. We have already discussed many advanced shadow mapping techniques which could provide a solution, especially the PCSS [Fernando 2005] technique, which would have the added benefit of providing a varied soft edge to the 3D shadow depending on the distance from the light and the occluding model. We believe this would give a varied penumbra to our 3D shadow which could be seen through the particle system, greatly improving the aesthetics of the overall effect. This would likely be a fruitful path for future research.

The soft volumetric particles technique could also be used to reduce the hard edge artefacts but would need to be modified to work with a 2D shadow map instead of with regular 3D geometry. This could be a path for future research.

Finally combining our method with a self-shadowing and an alpha-blending technique should enrich the final scene. A sorting technique will likely be needed to implement these and we suggest using the *odd-even merge sort* [Kipfer and Westermann 2005] because of its ability to perform sorting passes over multiple frames, gradually and smoothly moving towards a sorted data set. The *half-angle axis sorting* [Ikits et al. 2004] concept may also be useful in reducing the workload. Green [Green 2008] uses this technique successfully to implement a self-shadowing particle system. Deep opacity maps [Yuksel and Keyser 2008] look to be another boon for self-shadowing and might be

applied instead of a regular slicing technique, certainly worth investigating in future research.

6.3.2 Surface-Curvature Ambient Occlusion

For Surface-Curvature Ambient Occlusion we suggest future work include investigating using multiple adjacent vertices and possibly taking an average to improve the accuracy when calculating the local surface curvature. Also the method needs to be thoroughly tested on high and low polygon models which may affect the results.

Finally, future work could look at using a normal map when calculating the surface curvature to improve the accuracy of the results and explore how this method could fit in with level of detail techniques.

Appendix A

The following 2 pages are a short poster paper published at the 2010 Eurographics conference [Bass and Anderson 2010].

Eurographics page 1

Eurographics page 2

Glossary

API (application programming interface) – an interface implemented by software allowing other software to interact via commands. Offers a layer of abstraction between software so that higher-level programs can become hardware independent. Allows for high-level software commands to interact with low-level software and hardware without requiring detailed knowledge of the low-level software.

Billboard – a camera aligned rectangular primitive (often built from two triangles) with an attached texture also known as a sprite or an imposter under some circumstances.

Billboarding – a rendering technique involving the use of billboards (see above).

GPU (graphics processing unit) – a parallel processor designed for the acceleration of 3D graphics rendering.

MRT (multiple render target) – the ability to render out to multiple different render targets in one pass.

OpenGL (open graphics library) – an open-source cross-platform graphics API providing programmers with many graphics related commands providing a basic foundation for applications using accelerated graphics.

Pixel (picture element) – a fundamental element (building block) of a raster image, which contains a solid colour, made up from RGB channels.

QUICK (quadratic upstream interpolation for convective kinematics) – an advection technique which can be used to simulate the movement of fluid.

SDK (software development kit) – a group of development tools and reference material which aid a software developer in writing programs for a specific software platform.

Shadow volumes – a shadowing technique that involves extruding a shadow volume from the occluding mesh which encapsulates the shadowed parts of the scene.

Shadow mapping – a shadowing technique that involves creating a height map of the occluding mesh from the light position. The height map is then used to determine if a pixel is in shadow or not by comparing the depth of the pixel and the depth in the height map.

SCAO (surface-curvature ambient occlusion) – a technique for estimating the ambient light reaching a surface by using the geometry shader to calculate the local surface curvature.

SSAO (screen space ambient occlusion) – a post process lighting effect that adds an estimate of the amount of ambient lighting in a scene by using the depth of the pixels in screen space.

Texel (texture pixel) – a pixel belonging to a texture.

Voxel (volumetric pixel) – a fundamental element (building block) of a volume split into a regular grid of small equally sized cubes.

References

Akenine-Moller, T. and Assarsson, U. (2002) 'Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges'. pp. 297-306 *Eurographics Workshop on Rendering*: Eurographics ISBN: 1-58113-534-3

Annen, T.; Mertens, T.; Bekaert, P.; Seidel, H.-P. and Kautz, J. (2007) 'Convolution Shadow Maps'. pp. 51-60 *Rendering Techniques 2007: Eurographics Symposium on Rendering*: Eurographics 18 ISBN: 978-3-905673-52-4

Annen, T.; Mertens, T.; Seidel, H.-P.; Flerackers, E. and Kautz, J. (2008) 'Exponential shadow maps'. pp. 155-161 *Graphics Interface*: Canadian Information Processing Society ISBN: 978-1-56881-423-0

Ashikhmin, M. and Shirley, P. (2000) 'An Anisotropic Phong Light Reflection Model'.: The University of Utah

Assarsson, U.; Dougherty, M.; Mounier, M. and Akenine-Möller, T. (2003) 'An Optimized Soft Shadow Volume Algorithm with Real-Time Performance'. pp. 33-40 *Graphics Hardware*: Eurographics ISBN: 1-58113-739-7

Bass, C. J. and Anderson, E. F. (2010) 'Real-Time Smoke Rendering and Light Interaction'. *Eurographics Posters*: Eurographics

Bavoil, L. (2008) 'Percentage-Closer Soft Shadows'.: NVidia [online] available from <http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/PercentageCloserSoftShadows/doc/PercentageCloserSoftShadows.pdf>

Bavoil, L. and Sainz, M. (2008) 'Screen Space Ambient Occlusion'.: NVidia [online] available from <http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/ScreenSpaceAO/doc/ScreenSpaceAO.pdf>

Bilodeau, B. and Songy, M. (1999) 'Real Time Shadows'. *Creativity*: Creative Labs Inc.

Blinn, J. F. (1977) 'Models of Light Reflection for Computer Synthesized Pictures'. pp. 192-198 *SIGGRAPH*: ACM 11 (2)

Blythe, D. (2006) 'The Direct3D 10 System'. pp. 724-734 *SIGGRAPH*: ACM 25 (3) ISBN: 1-59593-364-6

Bunnell, M. (2005) 'Dynamic Ambient Occlusion and Indirect Lighting'. ch. 14 pp. 223-233 NVidia *GPU Gems 2: Programming Techniques for High-Performance*

Graphics and General-Purpose Computation: Addison-Wesley ISBN: 0-321-33559-7

Bunnell, M. and Pellacini, F. (2004) 'Shadow Map Antialiasing'. ch. 11 NVidia *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*: Addison-Wesley ISBN: 0-321-22832-4

Cantlay, I. (2007) 'High-Speed, Off-Screen Particles'. ch. 23 pp. 513-529 NVidia *GPU Gems 3*: Addison-Wesley ISBN: 0-321-51526-9

Carmack, J. (2000) Unpublished: 'Carmack on Shadow Volumes'. [online] available from <http://developer.nvidia.com/attach/6832>

Chan, E. and Durand, F. (2003) 'Rendering Fake Soft Shadows with Smoothies'. pp. 208-218 *Eurographics Workshop on Rendering*: Eurographics ISBN: 3-905673-03-7

Cohen, J. M.; Tariq, S. and Green, S. (2010) 'Interactive Fluid-Particle Simulation using Translating Eulerian Grids'. pp. 15-22 *SIGGRAPH*: ACM ISBN: 978-1-60558-939-8

Cook, R. L. and Torrance, K. E. (1982) 'A Reflectance Model for Computer Graphics'. pp. 7-24 *TOG*: ACM 1 (1)

Crane, K.; Llamas, I. and Tariq, S. (2007) 'Real-Time Simulation and Rendering of 3D Fluids'. ch. 30 pp. 633-675 NVidia *GPU Gems 3*: Addison-Wesley ISBN: 0-321-51526-9

Crow, F. C. (1977) 'Shadow Algorithms for Computer Graphics'. pp. 242-248 *SIGGRAPH*: ACM 11 (2)

Dimitrov, R. (2007) 'Cascaded Shadow Maps'.: NVidia [online] available from <http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf>

direct to video (2009) 'A Thoroughly Modern Particle System'.: direct to video [online] available from <http://directtovideo.wordpress.com/2009/10/06/a-thoroughly-modern-particle-system>

Donnelly, W. and Demers, J. (2004) 'Generating Soft Shadows Using Occlusion Interval Maps'. ch. 13 NVidia *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*: Addison-Wesley ISBN: 0-321-22832-4

Donnelly, W. and Lauritzen, A. (2006) 'Variance Shadow Maps'. pp. 161-165 *Symposium on Interactive 3D Graphics*: ACM ISBN: 1-59593-295-X

Everitt, C. and Kilgard, M. J. (2002) 'Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering'.: NVidia [online] available from <http://developer.nvidia.com/attach/6831>

Fedkiw, R.; Stam, J. and Jensen, H. W. (2001) 'Visual Simulation of Smoke'. pp. 15-22 *SIGGRAPH*: ACM ISBN: 1-58113-374-X

Fernando, R. (2005) 'Percentage-Closer Soft Shadows'. p. 35 *SIGGRAPH*: ACM

Fillion, D. and McNaughton, R. (2008) 'Starcraft 2 Effects and Techniques'. pp. 133-164 *SIGGRAPH*: ACM

Fraedrich, R.; Auer, S. and Westermann, R. (2010) 'Efficient High-Quality Volume Rendering of SPH Data'. *IEEE Visualization*: IEEE 16 (6)

Green, S. (2008) 'Volumetric Particle Shadows'.: NVidia [online] available from <http://developer.download.nvidia.com/compute/cuda/sdk/website/C/src/smokeParticles/doc/smokeParticles.pdf>

Hoberock, J. and Jia, Y. (2007) 'High-Quality Ambient Occlusion'. ch. 12 pp. 257-274 NVidia *GPU Gems 3*: Addison-Wesley ISBN: 0-321-51526-9

Howes, L. and Thomas, D. (2007) 'Efficient Random Number Generation and Application Using CUDA'. ch. 37 pp. 805-831 NVidia *GPU Gems 3*: Addison-Wesley ISBN: 0-321-51526-9

Ikits, M.; Kniss, J.; Lefohn A. and Hansen C. (2004) 'Volume Rendering Techniques'. ch. 39 NVidia *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*: Addison-Wesley ISBN: 0-321-22832-4

Jenson, H. W. (1996) 'Global Illumination using Photon Maps'. pp. 21-30 *Eurographics*: Springer-Verlag ISBN: 3-211-82883-4

Khronos Group (2006) 'OpenGL 2.1'. [online] available from <http://www.opengl.org/>

Kim, T.-Y. and Neumann, U. (2001) 'Opacity Shadow Maps'. pp. 177-182 *Eurographics*: Springer-Verlag ISBN: 3-211-83709-4

Kipfer, P. and Westermann, R. (2005) 'Improved GPU Sorting'. ch. 46 pp. 733-747 NVidia *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*: Addison-Wesley ISBN: 0-321-33559-7

Kolb, A.; Latta, L. and Rezk-Salama, C. (2004) 'Hardware-based Simulation and Collision Detection for Large Particle Systems'. pp. 123-131 *SIGGRAPH*: ACM ISBN: 3-905673-15-0

Kruger, J. and Westermann, R. (2003) 'Acceleration Techniques for GPU-based Volume Rendering'. p. 38 *IEEE Visualization*: IEEE Computer Society ISBN: 0-7695-2030-8

Latta, L. (2004) 'Building a Million Particle System'.: Massive Development GmbH [online] available from <http://www.2ld.de/gdc2004/MegaParticlesPaper.pdf>

Latta, L. (2007) 'Everything about Particle Effects'.: Game Developers Conference [online] available from <http://www.2ld.de/gdc2007/EverythingAboutParticleEffectsSlides.pdf>

Lauritzen, A. (2007) 'Summed-Area Variance Shadow Maps'. ch. 8 pp. 157-183 NVidia *GPU Gems 3*: Addison-Wesley ISBN: 0-321-51526-9

Lauritzen, A. and McCool, M. (2008) 'Layered Variance Shadow Maps'. pp. 139-146 *Graphics Interface*: Canadian Information Processing Society ISBN: 978-1-56881-423-0

Lokovic, T. and Veach, E. (2000) 'Deep Shadow Maps'. pp. 385-392 *SIGGRAPH*: ACM ISBN: 1-58113-208-5

Lorach, T. (2007) 'Soft Particles'.: NVidia [online] available from <http://developer.download.nvidia.com/whitepapers/2007/SDK10/SoftParticles_hi.pdf>

Microsoft (2010) 'DirectX SDK June 2010'. [online] available from <http://msdn.microsoft.com/en-us/directx/default.aspx>

Mittring, M. (2007) 'Finding Next Gen: CryEngine 2'. pp. 97-121 *SIGGRAPH*: ACM

Molemaker, J.; Cohen, J. M.; Patel, S. and Noh, J. (2008) 'Low Viscosity Flow Simulations for Animation'. pp. 9-18 *Symposium on Computer Animation*: Eurographics ISBN: 978-3-905674-10-1

Muller, M.; Charypar, D. and Gross, M. (2003) 'Particle-Based Fluid Simulation for Interactive Applications'. pp. 154-159 *Symposium on Computer Animation*: Eurographics ISBN: 1-58113-659-5

Myers, K. (2007) 'Variance Shadow Mapping'.: NVidia [online] available from <http://developer.download.nvidia.com/whitepapers/2007/SDK10/VarianceShadowMapping.pdf>

Myers, K.; Fernando, R. and Bavoil, L. (2008) 'Integrating Realistic Soft Shadows into Your Game Engine'.: NVidia [online] available from <http://developer.download.nvidia.com/whitepapers/2008/PCSS_Integration.pdf>

NVidia (2007) 'NVidia CUDA SDK and GPU Computing'. [online] available from <http://developer.nvidia.com/object/gpucomputing.html>

NVidia (2007) 'NVidia Direct3D and OpenGL SDKs version 10'. [online] available from <http://developer.nvidia.com/object/sdk_home>

Oren, M. and Nayar, S. K. (1994) 'Generalization of Lambert's Reflectance Model'. pp. 239-246 *SIGGRAPH*: ACM ISBN: 0-89791-667-0

Perlin, K. and Hoffert, E. M. (1989) 'Hypertexture'. pp. 253-262 *SIGGRAPH*: ACM 23 (3) ISBN: 0-89791-312-4

Pharr, M. and Green, S. (2004) 'Ambient Occlusion'. ch. 17 pp. 279-292 NVidia *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*: Addison-Wesley ISBN: 0-321-22832-4

Phong, B. T. (1973) 'Illumination of Computer-generated Images'.: The University of Utah

Porter, T. and Duff, T. (1984) 'Compositing Digital Images'. pp. 253-259 *SIGGRAPH*: ACM 18 (3) ISBN: 0-89791-138-5

Reeves, W. T. (1983) 'Particle Systems - A Technique for Modeling a Class of Fuzzy Objects'. pp. 359-375 *SIGGRAPH*: ACM ISBN: 0-89791-109-1

Reeves, W. T.; Salesin, D. H. and Cook, R. L. (1987) 'Rendering Antialiased Shadows with Depth Maps'. pp. 283-291 *SIGGRAPH*: ACM 21 (4) ISBN: 0-89791-227-6

Reynolds, C. W. (1987) 'Flocks, Herds and Schools: A Distributed Behavioral Model'. pp. 25-34 *SIGGRAPH*: ACM ISBN: 0-89791-227-6

Scherzer, D.; Jeschke, S. and Wimmer, M. (2007) 'Pixel-Correct Shadow Maps with Temporal Reprojection andShadow Test Confidence'. *Symposium on Rendering*: Eurographics

Selle, A.; Fedkiw, R.; Kim, B.; Liu, Y. and Rossignac, J. (2008) 'An Unconditionally Stable MacCormack Method'. pp. 350-371 *Journal of Scientific Computing*: Plenum Press 35 (2-3)

Selle, A.; Mohr, A. and Chenney, S. (2004) 'Cartoon Rendering of Smoke Animations'. pp. 57-60 *Non-Photorealistic Animation and Rendering*: ACM ISBN: 1-58113-887-3

Sintorn, E. and Assarsson, U. (2007) 'Fast Parallel GPU-Sorting Using a Hybrid Algorithm'. pp. 1381-1388 *Journal of Parallel and Distributed Computing*: Academic Press Inc. 68 (10)

Stam, J. (1999) 'Stable Fluids'. pp. 121-128 *SIGGRAPH*: ACM ISBN: 0-201-48560-5

Stam, J. (2003) 'Real-Time Fluid Dynamics for Games'.: Game Developers Conference

Stich, M.; Wachter, C. and Keller, A. (2007) 'Efficient and Robust Shadow Volumes Using Hierarchical Occlusion Culling and Geometry Shaders'. ch. 11 pp. 239-257 NVidia *GPU Gems 3*: Addison-Wesley ISBN: 0-321-51526-9

Strauss, P. S. (1990) 'A Realistic Lighting Model for Computer Animators'. pp. 56-64 *IEEE Computer Graphics and Applications*: IEEE Computer Society Press 10 (6)

The Valve Developer Community (2010) 'Particle System Overview'.: The Valve Developer Community [online] available from <http://developer.valvesoftware.com/wiki/Particle_System_Overview>

van Pelt, R.; Vilanova i Bartroli, A. and van de Wetering, H. (2010) 'Illustrative Volume Visualization using GPU-Based Particle Systems'. pp. 571-582 *IEEE Visualization*: IEEE 16 (4)

Ward, G. (1992) 'Measuring and Modelling Anisotropic Reflection'. pp. 265-272 *SIGGRAPH*: ACM 26 (2)

Williams, L. (1978) 'Casting Curved Shadows on Curved Surfaces'. pp. 270-274 *SIGGRAPH*: ACM 12 (3)

Wyman, C. and Hansen, C. (2003) 'Penumbra Maps: Approximate Soft Shadows in Real-Time'. pp. 202-207 *Eurographics Workshop on Rendering*: Eurographics ISBN: 3-905673-03-7

Yuksel, C. and Keyser, J. (2008) 'Deep Opacity Maps'. *Computer Graphics Forum*: Eurographics 27 2

Zhang, F.; Sun, H. and Nyman, O. (2007) 'Parallel-Split Shadow Maps on Programmable GPUs'. ch. 10 pp. 203-235 NVidia *GPU Gems 3*: Addison-Wesley ISBN: 0-321-51526-9