Performance of Deferred and Forward Shading under Different Lighting Conditions

Alexandr Polisciuc

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the Degree of

Master of Science
in
Computer Engineering

Eastern Mediterranean University
July 2013
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Prof. Dr. Elvan Yılmaz
Director

I certify that this thesis satisfies the requirements as a thesis for the degree of Master of Science in Computer Engineering.

Assoc. Prof. Dr. Muhammed Salamah
Chair, Department of Computer Engineering

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Master of Science in Computer Engineering.

Prof. Dr. Hakan Altınçay
Supervisor

Examining Committee

1. Prof. Dr. Hakan Altınçay _____

2. Prof. Dr. Hasan Kömürcügil _____

3. Asst. Prof. Dr. Ahmet Ünveren _____

ABSTRACT

The field of 3D computer graphics deals with ways of generating 2D images from 3D scene representations. This process is called rendering and its performance is one of the central problems in the field. Understanding performance implications of 3D graphics algorithms and testing them in different scenarios enables professionals in game, film, scientific and military industries to make informed decisions on which algorithms are best suited for their problem. In this thesis a close look was taken at performance of two most popular rendering approaches in real-time 3D graphics – deferred shading and forward shading. We investigated four different scenarios: *many small* lights, *many big* lights, *many big* lights *with shadows* and a mixed case of *many small* lights along with *several big shadow-casting* lights. Deferred shading showed better performance than forward shading in all tests, with the biggest gain obtained in case of having high numbers of small lights. When shadow-casting lights were present, the difference in performance, although significant, was not as hugely different as in case of small lights alone.

Keywords: 3D Computer Graphics, Rendering, Deferred Shading, Forward Shading

ÖZ

3B bilgisayar grafik çalışmaları, 3B görüntülerden 2B imgeler üretme yöntemleri üzerindedir. Bu işlem imge oluşturma olarak bilinmekte ve başarımı bu alandaki esas problemlerden birisi olarak kabul edilmektedir. 3B grafik algoritmalarının başarımlarının etkilerini anlamak ve farklı senaryolar için onları test etmek, oyun, film, bilimsel ve askeri endüstrilerde hangi algoritmanın en uygun olacağı konusunda daha bilinçli karar almaya olanak sağlamaktadır. Bu tezde gerçek zamanlı 3B grafik alanında iki popüler imge oluşturma yöntemi - erteleme tabanlı gölgelendirme ve ileri gölgelendirme yakından incelenmiştir. Dört farklı senaryo üzerinde çalışılmıştır: çoklu küçük ışık, çoklu büyük ışık, gölgeli çoklu büyük ışık ve çoklu küçük ışık ile birkaç gölge oluşturan ışık karışımı. Erteleme tabanlı gölgelendirme, tüm testlerde ileri gölgelendirme yöntemine göre daha iyi başarım göstermiş, en yüksek kazanımı da çoklu küçük ışık durumunda sağlamıştır. Gölge oluşturan ışıkların olduğu durumda, belirgin bir başarım farkı olmakla birlikte yalnızca küçük ışıkların kullanıldığı durumdaki gibi büyük olmamıştır.

Anahtar Kelimeler: 3B Bilgisayar Grafiği, Imge Oluşturma, Erteleme Tabanlı Gölgelendirme, Ileri Gölgelendirme

ACKNOWLEDGMENTS

I would like to thank my supervisor, Prof. Altınçay, for letting me pursue my interest in real-time rendering. At the risk of sounding banal, I will say that this brought me one big step closer to the vocation I have always dreamed to follow.

I am very grateful to my Mom and Dad, whose confidence in me and moral support regularly boosted my own self-confidence and optimism. They help me see the forest for the trees, even when sometimes all I concentrate on are the twigs and leaves.

A big thank you also goes to my friend Aram, who would lean over my shoulder and tell me, much to my surprise, that what I was doing looked “cool” and “awesome”. That was something I really needed to keep me going from day to day.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

This work is concerned with 3D graphics and focuses specifically on performance characteristics of *forward* and *deferred* shading, which are the two most popular rendering approaches in real-time graphics applications.

The term *rendering* means producing a 2D image of a 3D scene. A 3D scene consists of light sources and objects. *Shading* is the process of calculating an object's illumination due to light sources. Transforming geometric primitives (typically triangles) of objects is part of the rendering process and is done before shading.

Forward shading is the straightforward (and thus default) way of rendering, in which shading of an object is done immediately after its geometric primitives are transformed. Performance of this approach does not scale well as number of light sources in a 3D scene increases.

Deferred shading (Saito and Takahashi, 1990) defers the shading operation, separating the rendering process into the geometry pass (transformation of geometric primitives) and the shading pass. It has grown in popularity, substituting the standard forward approach in many high-profile engines. The motivation for this move is the fact that deferred approach allows cheaper rendering of high number of light sources. Most previous publications, such as (Hargreaves and Harris, 2004), (Shishkovtsov, 2005), (Filion and McNaughton, 2008) and (Koonce, 2008), discuss implementation details, limitations, trade-offs and optimizations, reporting increase in performance, but avoiding detailed analysis. More performance-centric works, such as (Postma,

2009), (Hoef and Zalmstra, 2010), (Lauritzen, 2010) and (Olsson, 2010), focus on limited cases only.

The goal of this work is to conduct a case study and look at some conditions not mentioned in literature. We look at four different scenarios and see how performance scales in each of the following:

- many *small* area of influence light sources *without shadows*
- many *large* area of influence light sources *without shadows*
- many *large* area of influence *shadow-casting* light sources
- some *large shadow-casting* light sources and many *small non-shadow-casting* light sources

While first two scenarios are the most popular to look at in literature, the third and fourth ones are, to the best of our knowledge, never seen to be investigated, even though the mixed case of *many small and several big lights* is most representative of real-world situations. This work is intended to fill this gap.

Chapter 2 of this work covers the fundamentals of computer graphics, describing the components constituting a 3D scene, the mathematical apparatus for producing a 2D image of a 3D scene and the modern graphics processing unit (GPU) pipeline. After dealing with fundamentals, the chapter covers the concepts of forward shading and deferred shading. Chapter 3 describes the experiment setup, providing descriptions of both forward and deferred shading renderers' implementations. The test scene, test runs' specifics and hardware used are specified in this chapter as well. Chapter 4 provides the results of the experiments done and analysis of the two methods' observed behaviors in different tests. Chapter 5 offers conclusions and outlines future work.

Chapter 2

FUNDAMENTALS AND CONCEPTS OF COMPUTER GRAPHICS

2.1 Computer Graphics and Its Applications

The central problem of 3D computer graphics is creation of a 2D image out of a mathematical description of a 3D scene. This is an important problem for many fields of human activity, because humans rely on visualization for conveying and consuming information. For instance, in physics, a physical process is better understood when one sees a 3D graphics simulation of the process. On the other hand, computer aided design (CAD), would be impossible without 3D computer graphics. 3D computer graphics is also used in military simulations (Figure 2.1), automotive crash tests, film and computer games industries. By now, it is a mature discipline and is a separate branch of Computer Science. We will refer to 3D computer graphics simply as *graphics* throughout this text.

Figure 2.1: Military F-15 Jet Simulator
Source: www.ign.com/blogs/jeydt/2012/12/14/feature-press-x-to-kill-the-relationship-between-video-games-and-the-military

It is common to separate offline and real-time graphics. *Offline graphics* refers to methods of generating imagery at non-interactive frame rates (below the threshold of 15 frames per second); whereas *real-time graphics* refers to methods that enable generation of images at frame-rates above that threshold. Real-time methods are used in cases where user input affects what is being drawn to the output device. In cases where user input does not affect the computer generated imagery, computation may be done offline and displayed at a later time.

Film industry is a perfect example where offline graphics makes sense. Since the viewer has no agency over what happens on the screen, imagery may be generated beforehand. This allows use of scenes and visual effects of very high complexity. Generating a single frame even on a cluster of computers may take a couple of hours, but it is still acceptable, because the constraint on interactivity is lifted.

Real-time methods are used in CAD applications, military simulators, training simulators in automotive and airspace industries, as well as in the computer games industry, where visual output depends on user input. The majority of research and

innovation is happening within the computer games industry. This is not surprising, since computer games industry has been reported to have become bigger than film industry and continues to grow. As the industry grows it demands better solutions and attracts talent to find those solutions.

As hardware becomes faster, some methods from the offline domain move into the real-time domain. This might give a wrong impression that all techniques in real-time graphics are simply borrowed from offline graphics, once they become feasible. Professionals in real-time graphics still need solutions that work faster because of tight performance constraints. They tend to optimize heavily and come up with approaches that would not be pursued by those working in offline graphics.

2.2 3D Graphics Concepts

In order to understand the problem explored in this work, familiarity with the process of producing a 2D image of a 3D scene and how graphics hardware operates is needed. The following subsections will describe how a 3D scene is represented, how a frame is generated from this information about the scene, how this process maps to hardware and the job of a renderer.

2.2.1 Scene Representation

A scene consists of *objects*, *light sources* and a *camera*. Objects are the 3D entities needed to be displayed, and lights (short for light sources) are the entities that make those objects visible in the first place. A scene must have at least one source of light. Otherwise, none of the objects in the scene would be visible. Conversely, if there are no objects in the world, there is nothing to see even in the presence of

thousands of lights, because there is nothing to illuminate. An observation point has to be present in the scene in order to see something, which is what a camera is for.

In graphics applications, representation and storage of objects and lights in a computer is an important problem. For objects, the mainstream approach is to represent them as an array of 3D space points. Each point is a 3-tuple containing Cartesian coordinates (x, y, z), and is called a *vertex*. Vertices comprise triangles which represent the visible surface of an object. Figure 2.2 depicts an object, with the first view showing only points, the second view showing all the edges of triangles and the third one showing the illuminated object.

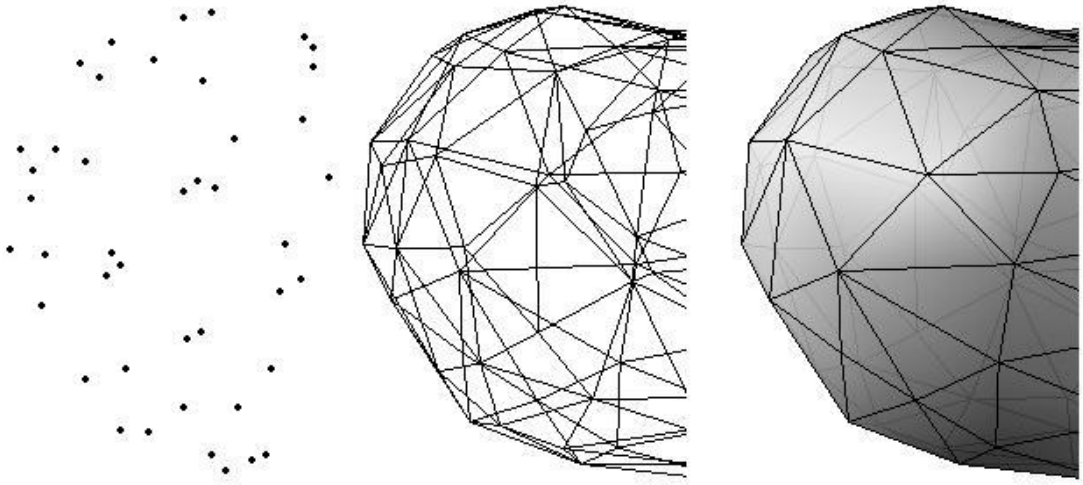

Figure 2.2: Vertices, Triangles and Lit Surface of an Object
Source: www.magic.ubc.ca/artisynth/pmwiki.php?n=OPAL.MarkoMarjanovic

As for lights, since there are several types of light sources, each of them is represented by distinct attributes. The four most common types are *ambient*, *directional*, *point* and *spot* lights. Figure 2.3 shows conceptual pictures for three and Figure 2.4 shows all four in action.

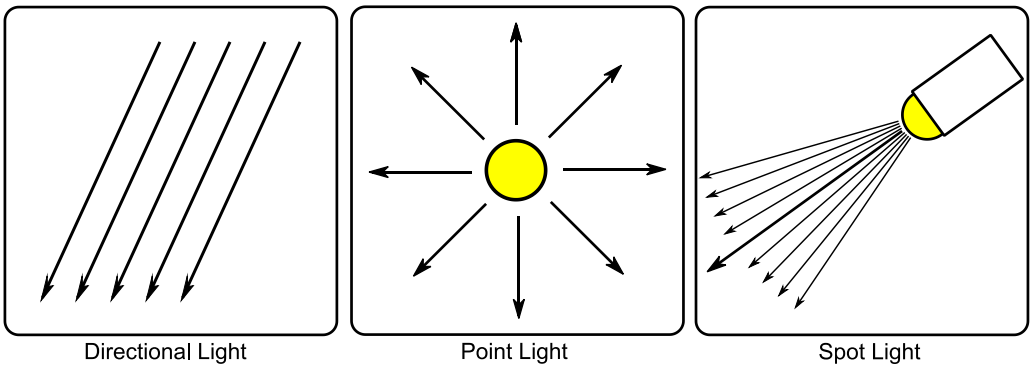

Figure 2.3: Types of Light Sources

An ambient light is a simplified approach to simulate indirect illumination, whereby surfaces receive light even if they do not face the light source. This can be represented by a single value – the amount of light the whole scene receives. This yields a uniformly lit scene, which lacks any feeling of depth, as can be seen in Figure 2.4.

Directional light is a representation of sunlight or moonlight. This type of light has the same direction at any point in a given scene. Since the source itself is so far away from the objects in the scene, the light is considered to be coming from an infinite distance. Hence the fixed direction and intensity should be stored for representing such a source.

A point light is akin to a candle or light bulb, where light emanates from a certain point in space in all directions around it. Defining such a light requires specification of the origin position and its intensity.

A spot light mimics behavior of a flashlight. In this case a cone of light is shining from a certain point in space. Representation of a spot light requires the light's position, direction and the angle of the bounding cone to be specified.

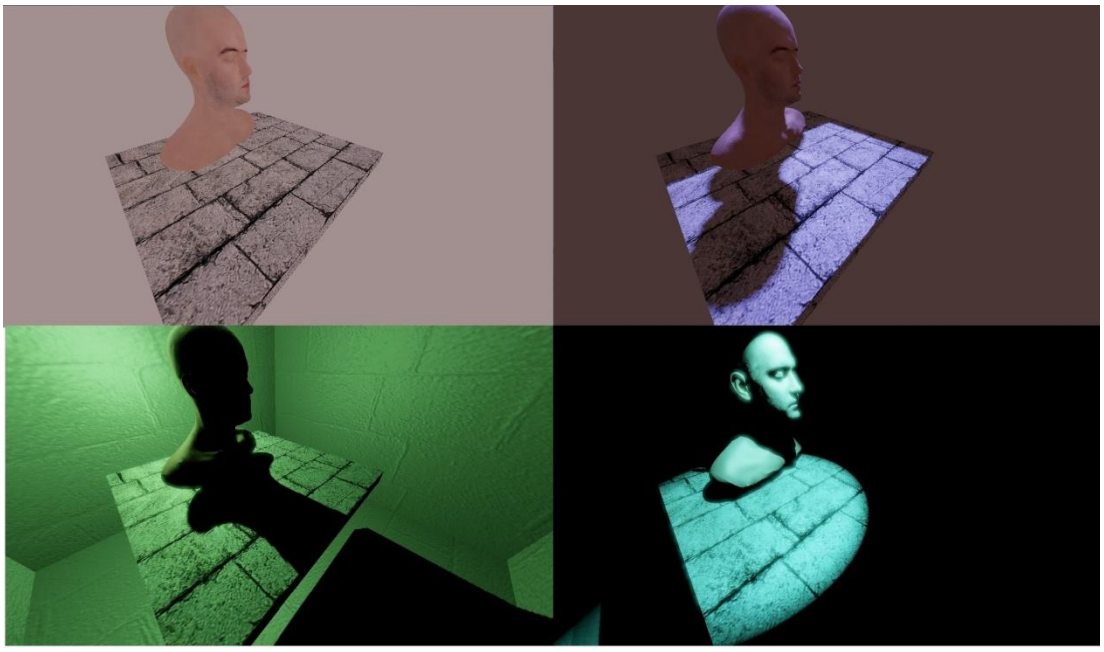

Figure 2.4: Illumination Effects of Different Light Sources. Left Top to Right Bottom: Ambient Light, Directional Light, Point Light and Spot Light

A camera representation needs to store at least five parameters: camera's position, direction of view, field of view (FOV), near plane and far plane. These parameters define a frustum (a pyramid with a chopped off top) as shown in Figure 2.5. Everything located within this frustum will be visible. Everything outside will be clipped and not visible. FOV is an angle which defines how wide the region of view is. Far plane bounds the maximum distance of vision. Near plane is the bound on closest objects that are visible. Visible parts of the scene are projected onto the near plane.

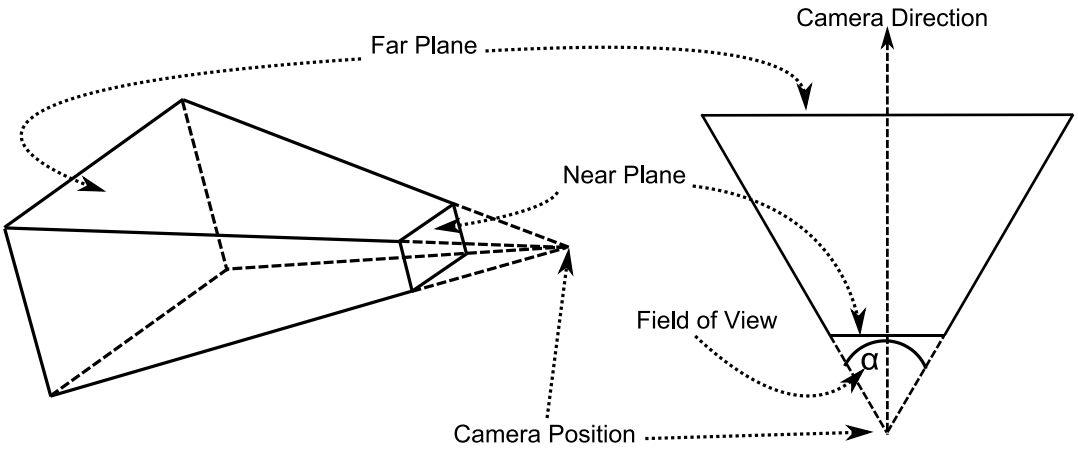

Figure 2.5: Camera Frustum

Having a number of objects (instances of object representations described above), a number of lights (instances of the four types of light) and a camera means having defined a scene. This information is enough to generate a sequence of images called *frames*, which are displayed to the user. Next section describes how the above mentioned scene data is used to construct a frame.

2.2.2 Generation of a Frame

A frame is essentially an image that contains projection of a scene onto a surface. In 3D graphics the principle is very similar to the way human eye or a camera works. However we are no longer dealing with the physical process – there are no photons hitting a retina of an eye or a matrix of a camera. This process needs to be simulated with mathematical operations. In fact, 3D graphics is all about coming up with a mathematical model for human vision and efficiently executing it on hardware.

Taking an object as a collection of vertices in 3D space and making it visible on a flat surface is achieved by applying a series of transformations and then rasterizing the triangles. *Rasterization* means turning 3D representation of an object into a 2D image. That is, having three vertices of a triangle it needs to be decided which pixels

the triangle covers. Before applying rasterization, coordinates of the object's vertices need to be transformed into a suitable space, which is convenient for performing rasterization.

2.2.2.1 Transformations

Change of position and change of orientation can be represented by a translation operation and a rotation operation, respectively. Translation and rotation operations on vectors in computer graphics are applied using matrices, as described below. To specify a rotation in 3D space a 3x3 matrix $\mathbf{M}_{\text{rotation}}$ is sufficient:

$$\mathbf{M}_{\text{rotation}} = \begin{pmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{pmatrix} \tag{2.1}$$

where vectors $\mathbf{A}_{\text{xyz}}$, $\mathbf{B}_{\text{xyz}}$ and $\mathbf{C}_{\text{xyz}}$ represent the new coordinate basis (after rotation) defined in terms of the old basis (before rotation) and each basis vector is specified using three coordinates of the old basis (x, y, z). Multiplication of a 3D column vector $\mathbf{v}$ (which may represent, for example, a vertex or a normal) by a 3x3 rotation matrix $\mathbf{M}_{\text{rotation}}$ yields a rotated vector:

$$\begin{aligned} \mathbf{M}_{rotation}\mathbf{v} &= \begin{pmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \\ &= \begin{pmatrix} A_x x + A_y y + A_z z \\ B_x x + B_y y + B_z z \\ C_x x + C_y y + C_z z \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \end{aligned} \tag{2.2}$$

For example, an identity rotation matrix has no effect on a vector, leaving it unchanged:

$$\mathbf{M}_{identity}\mathbf{v} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1x + 0y + 0z \\ 0x + 1y + 0z \\ 0x + 0y + 1z \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{2.3}$$

To give another example, one of the useful rotations is the rotation around the *x*-axis, represented as (Vince, 2006, p. 77):

$$\mathbf{M}_{rotation} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix} \tag{2.4}$$

As a numerical example, a 30° rotation around the *x*-axis changes the 3D space position of a (4, 3, 2) vertex to (4, 1.61, 3.24):

$$\begin{gathered} \mathbf{M}_{rotation}\mathbf{v} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(30°) & -\sin(30°) \\ 0 & \sin(30°) & \cos(30°) \end{pmatrix} \begin{pmatrix} 4 \\ 3 \\ 2 \end{pmatrix} = \\ = \begin{pmatrix} 1 \cdot 4 + 0 \cdot 3 + 0 \cdot 2 \\ 0 \cdot 4 + 0.87 \cdot 3 - 0.5 \cdot 2 \\ 0 \cdot 4 + 0.5 \cdot 3 + 0.87 \cdot 2 \end{pmatrix} = \begin{pmatrix} 4 \\ 1.61 \\ 3.24 \end{pmatrix} \end{gathered} \tag{2.5}$$

A sequence of several rotations (each represented by a 3x3 matrix) can be represented by a single rotation matrix:

$$\mathbf{M}_1\mathbf{M}_2 \ldots \mathbf{M}_N = \mathbf{M}_{12\ldots N} \tag{2.6}$$

It is very important that a whole chain of transformations represented by N matrices can be condensed into a single resulting transformation represented by one

matrix. This way each vertex has to be multiplied by one matrix instead of having to be multiplied by N matrices. This saves a lot of space and computation time.

A translation operation can be performed by adding a translation vector $\mathbf{v}_{\text{translation}}$ (specifying displacement along the *xyz* axes) to a 3D vector **v**:

$$\mathbf{v}_{translation} + \mathbf{v} = \begin{pmatrix} T_x \\ T_y \\ T_z \end{pmatrix} + \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} T_x + x \\ T_y + y \\ T_z + z \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \tag{2.7}$$

For example, a translation vector (1, -2, 3) applied to a vertex (4, 3, 2) will result into the vertex (5, 1, 5).

However, this representation does not allow a series of transformations (some of them rotations and some of them translations) to be merged into a single matrix. Unlike rotation, translation cannot be represented by a 3x3 matrix. There is no component in a 3x3 matrix which, during multiplication of said matrix by a 3D vector, could be added into the resulting value without being multiplied by one of the three components of the vector. A translation can alternatively be represented in the following way:

$$x' = 1x + 0y + 0z + T_x \times 1 = x + T_x \tag{2.8}$$

$$y' = 0x + 1y + 0z + T_y \times 1 = y + T_y \tag{2.9}$$

$$z' = 0x + 0y + 1z + T_z \times 1 = z + T_z \tag{2.10}$$

This representation is equivalent to multiplying a 4D vector (with the last component being equal to 1) by a 4x3 matrix:

$$\mathbf{M}_{translation}\mathbf{v} = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \tag{2.11}$$

Merging N 4x3 translation matrices into one by multiplying them is not possible because their dimensionality is incompatible. Adding an extra row solves this problem:

$$\mathbf{M}_{translation} = \begin{pmatrix} 1 & 0 & 0 & T_z \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.12}$$

A similar modification of a 3x3 rotation matrix yields a 4x4 equivalent matrix:

$$\mathbf{M}_{rotation} = \begin{pmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{pmatrix} \equiv \begin{pmatrix} A_x & A_y & A_z & 0 \\ B_x & B_y & B_z & 0 \\ C_x & C_y & C_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.13}$$

Using 4x4 matrices allows a chain of rotation and translation 4x4 matrix transformations to be reduced to a single matrix by multiplying the 4x4 matrices in this series of transformations.

In order to use the 4x4 matrix convention to represent transformations, 4D vectors have to be used. The coordinates in which these operations are performed are called *4D homogeneous coordinates*. To bring a vertex from 3D Cartesian space into 4D homogeneous coordinates, a fourth coordinate (*w*) is added with a value of 1:

$$\begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix} \equiv \begin{pmatrix} x_c \\ y_c \\ z_c \\ 1 \end{pmatrix} = \begin{pmatrix} x_h \\ y_h \\ z_h \\ w_h \end{pmatrix} \tag{2.14}$$

After multiplying 4D homogeneous coordinates by matrices the fourth component may have a value other than 1. To bring 4D homogeneous coordinates back to 3D Cartesian coordinates (project from 4D space to 3D space), all four components have to be divided by the w component:

$$\begin{pmatrix} x_h \\ y_h \\ z_h \\ w_h \end{pmatrix} \equiv \begin{pmatrix} \frac{x_h}{w_h} \\ \frac{y_h}{w_h} \\ \frac{z_h}{w_h} \\ \frac{w_h}{w_h} \end{pmatrix} = \begin{pmatrix} \frac{x_h}{w_h} \\ \frac{y_h}{w_h} \\ \frac{z_h}{w_h} \\ 1 \end{pmatrix} \equiv \begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix} \tag{2.15}$$

Initially, vertex coordinates are specified in *object space* in which the object is modeled. The center of an object is usually at the origin of object space. Any object has position and orientation within world space. To bring object vertex coordinates from object space to *world space* each vertex coordinate has to be multiplied first by a rotation matrix and then by a translation matrix:

$$\begin{gathered} \mathbf{M}_{transformation} = \mathbf{M}_{translation}\mathbf{M}_{rotation} = \\ = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} A_x & A_y & A_z & 0 \\ B_x & B_y & B_z & 0 \\ C_x & C_y & C_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \\ = \begin{pmatrix} A_x & A_y & A_z & T_x \\ B_x & B_y & B_z & T_y \\ C_x & C_y & C_z & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{gathered} \tag{2.16}$$

Objects in a scene could be arranged hierarchically, i.e. an object could be the parent of a number of child objects. In this case, a child's transformation is specified with respect to the parent's frame of reference. Consequently, a transformation from

object space to world space would involve a chain of transformation matrices representing rotation and translation. This chain, as was mentioned earlier, would be reduced to a single 4x4 object-to-world space matrix.

To bring the object vertices from world space to *camera space* (also called *eye space*) appropriate rotation and translation matrices have to be applied to the world space coordinates. In camera space the object vertex coordinates have values relative to the camera's frame of reference – camera is at the origin and facing down the negative direction of *z*-axis (for a right-hand coordinate system).

All the previous transformations (object to world space and world space to camera space) are done prior to projection. They are necessary to bring all vertices to a coordinate system in which *perspective projection* can be done.

When a 3D shape is projected onto a plane the result is a 2D shape. This is exactly what is needed to produce a 2D image of a 3D object. If the *xy*-plane is agreed to be the projection plane, the projection could be done by discarding the *z*-coordinate, as shown in Figure 2.6.

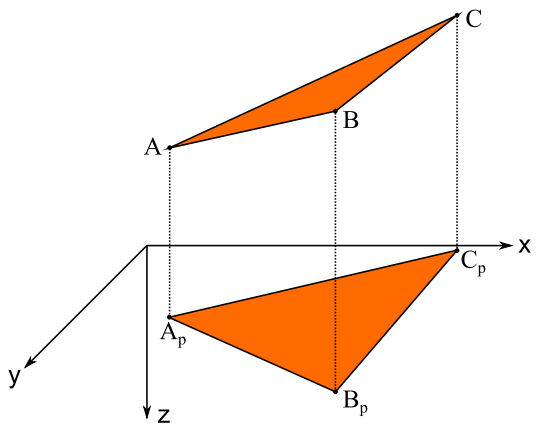

Figure 2.6: Projection of a Triangle

Taking object vertices in camera space and discarding the *z*-coordinate produces the *orthographic projection* of the object. This means that the object will look the

same size whether it is far away or close to the projection plane. To account for the fact that the farther away an object is, the less area it will occupy on the projection plane, camera space has to be mapped to another space in which projection can be done by discarding the *z*-component. The idea of mapping from camera space to a space that accounts for the perspective effect is illustrated in Figure 2.7.

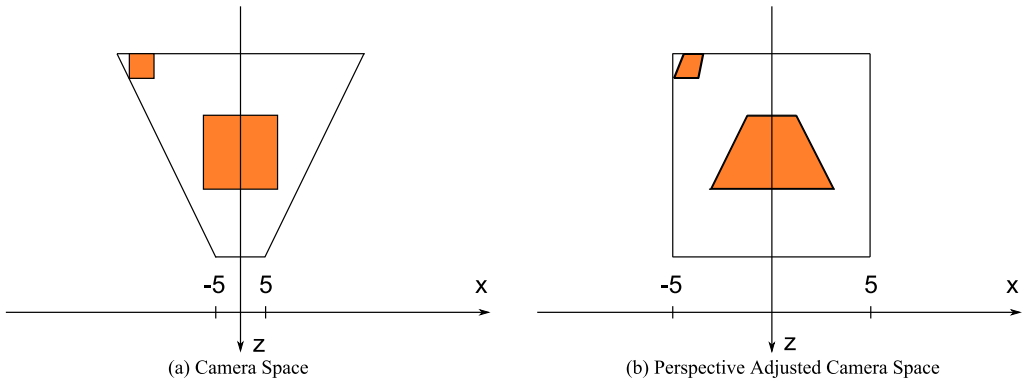

Figure 2.7: Perspective Effect Adjustment

The mapping from the previous example does not take place as a separate step. For convenience the view frustum is mapped to a unit cube, as shown in Figure 2.8. In this space, planes *x=1, x=-1, y=1, y=-1, z=1, z=-1* are now bounds of the view frustum. This is called *normalized device coordinates* (NDC) space. Coordinates are no longer 4D in NDC space, but rather in 3D Cartesian coordinates. That means perspective division of homogeneous coordinates happened. Right before this conversion, vertex coordinates were in c*lip space*. That is where clipping of triangles outside the camera frustum is performed. Magnitude of *xyz* components are checked to be less than magnitude of the *w* component, which is equivalent to the vertex being within the unit cube of NDC space.

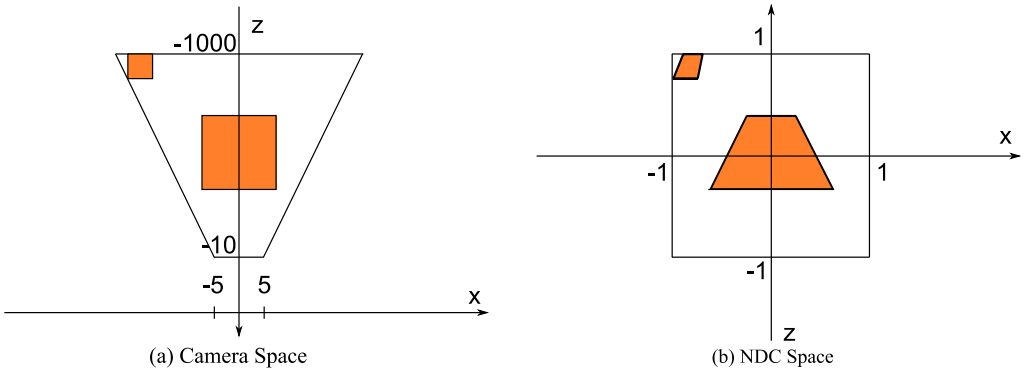

Figure 2.8: Camera Space to NDC Space Transformation

To get from *camera space* to *clip space*, vertex coordinates in camera space are multiplied by a projection matrix. This projection matrix is constructed from parameters of the camera (FOV, near plane, far plane) and the framebuffer (aspect ratio).

Before rasterization can take place, one last transformation is necessary – from NDC space to *window space*. The transformation applied is called viewport transformation. This operation maps xy-coordinates from (-1,1) range to (0, W_{max}) range; and z-coordinate is typically mapped from (-1,1) range to (0,1) range. W_{max} is the maximum window extent in x and y for corresponding coordinates. For example, with a resolution of 1600x1200 *x*-coordinate would be mapped to a range (0,1600) and *y*-coordinate to (0,1200). Figure 2.9 shows mapping to window resolution of 40 across X.

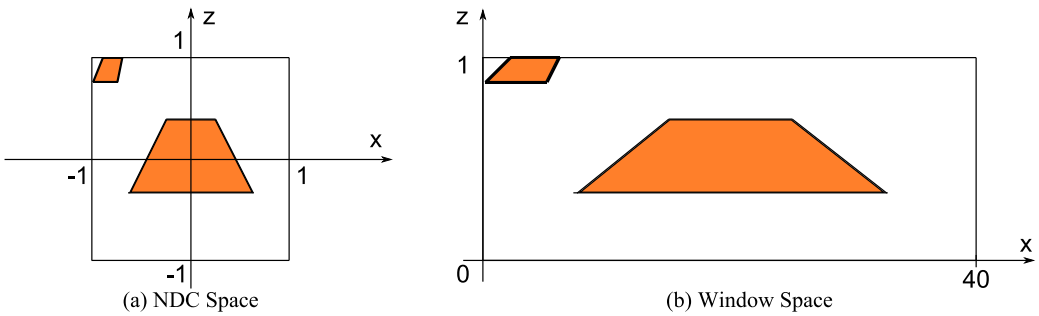

Figure 2.9: NDC Space to Window Space Transformation

2.2.2.2 Rasterization

After all work on geometry is done, triangles can be rasterized.

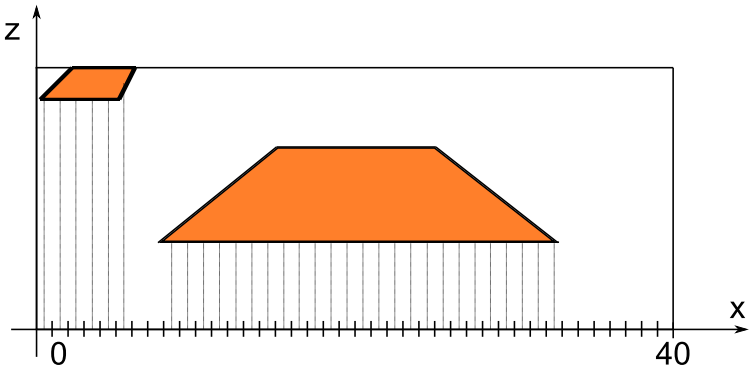

Figure 2.10: Rasterization

An image can be viewed as a grid where each cell is assigned a particular color. Having three vertices of a triangle in window space, it has to be decided which cells of the grid are covered by this triangle, as in Figure 2.11. To simplify, we assume the whole triangle is of one color.

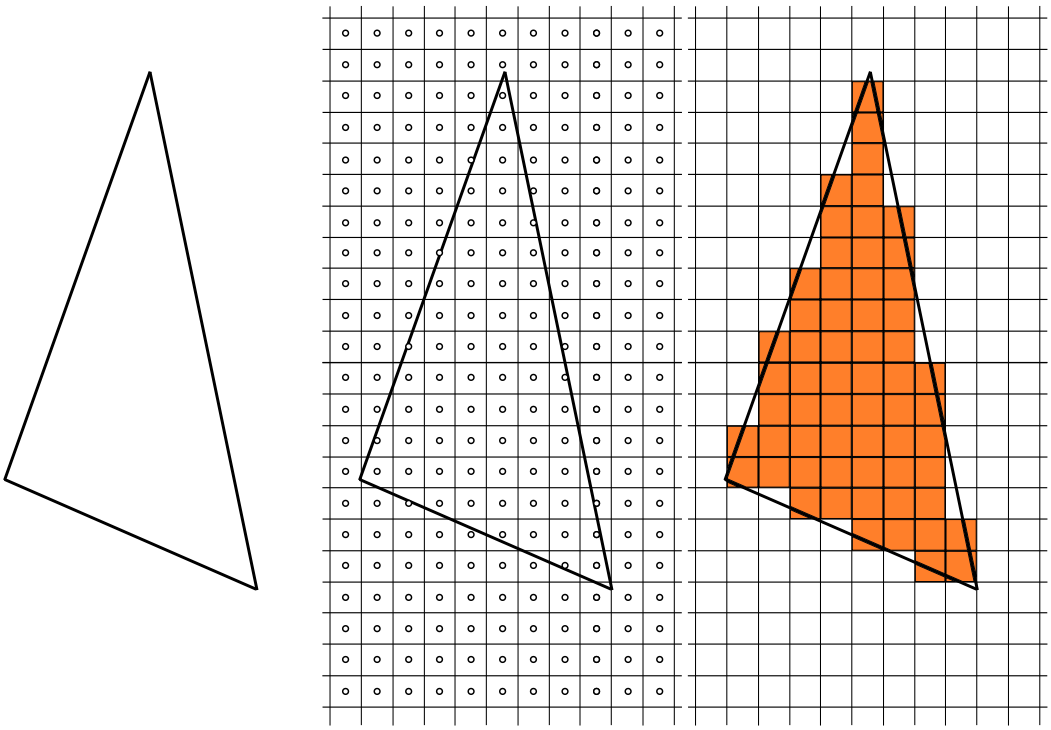

Figure 2.11: Rasterization of a Triangle

The cells of an image are called *pixels* (for picture element). A triangle may cover some part of a pixel but will not necessarily produce a pixel. A pixel is generated only if its center is covered by the triangle. On current mainstream hardware, graphics programmers do not have to implement rasterization by themselves. *Graphics processing units* (GPUs) run a scan conversion algorithm and produce pixels from geometric representation. This stage is an example of fixed-function operation. Hardware vendors choose the fastest way to implement scan conversion in hardware and a graphics programmer has no control over it.

Today, all computations from transformation to rasterization are done on the GPU. More than a decade ago, graphics chips had fixed function pipelines. This means that their operation was rigid and they were not programmable. Today, although some stages are still fixed to allow dedicated hardware perform a narrow

subset of tasks efficiently (i.e. rasterization, texture sampling etc.), there are many programmable stages. With the help of this flexibility a graphics programmer can run an arbitrary algorithm, for example, for simulating physical light transport. Future hardware is expected to become even more programmable; and many professionals believe that graphics processing may even move to the CPU entirely.

Next section gives a brief overview of what happens inside the GPU.

2.2.3 GPU Pipeline Overview

Is has to be mentioned that the modern GPU pipeline is very complicated. Here only a general and simplified overview is given. Figure 2.12 depicts the most important stages in the GPU pipeline.

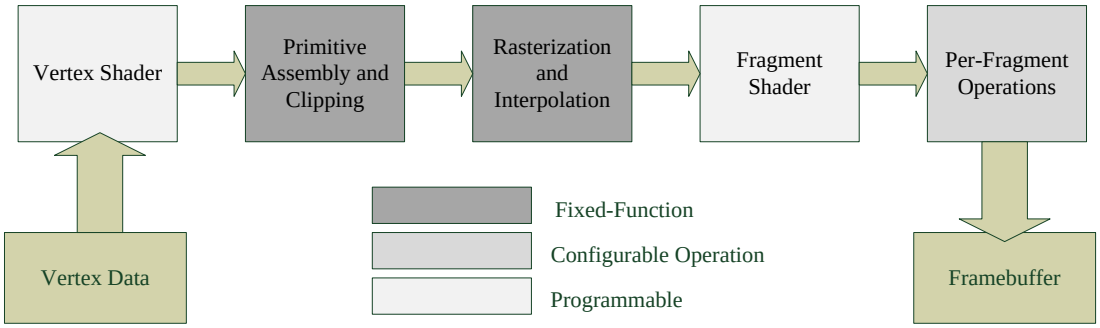

Figure 2.12: Simplified GPU Pipeline

The GPU pipeline consists of a mix of fixed-function and programmable stages. The core of the pipeline is the following chain: vertex processing, primitive assembly, clipping, rasterization, interpolation, fragment processing and per-fragment operations. Of these, only vertex processing and fragment processing are programmable, every other stage is either fixed-function or has configurable

behavior. Modern GPUs and graphics APIs provide some additional stages, which we do not consider.

The most interesting parts are obviously the programmable stages. All of them run *shaders*. A shader is a program written in a C-like language specifically developed for graphics. There are currently three shading languages for real-time applications. These languages are listed in Table 2.1. Shading languages are closely related to graphics APIs (OpenGL and Direct3D) which will be mentioned later.

Table 2.1: Shading Languages

	Full Name	API	Organization
GLSL	OpenGL Shading Language	OpenGL	Khronos
HLSL	High Level Shading Language	Direct3D	Microsoft
Cg	C for Graphics	OpenGL, Direct3D	Nvidia

We will consider only two types of shaders: *vertex* and *fragment.* A *vertex shader* is executed per vertex, i.e. all submitted vertices are fed as input into the vertex shader. A vertex shader typically transforms vertex coordinates to clip space. After that, the GPU combines vertices into triangles, discards those that lie outside the view frustum and clips those that lie on boundaries.

The following step is to rasterize surviving triangles – turn three vertices of a triangle into pixels. This is done by the fixed-function hardware in a process called scanline conversion, during which the GPU decides which pixels (picture elements of the final frame) are potentially covered by the triangle. This yields fragments – potential pixels. Each fragment will have vertex information interpolated (from vertex shader output) and available for each fragment in the fragment shader. Figure 2.13 shows a triangle defined by three vertices rasterized into many fragments. Each vertex has a different color value and those were interpolated across the whole

triangle for all fragments. For example, fragments around the center of a triangle have roughly the same percentage of red, blue and green in them.

Interpolated values do not have to be colors. They may be vectors indicating direction of normals of triangle's surfaces, they could be texture coordinates or any other piece of data that is specified per-vertex but will be needed per-fragment.

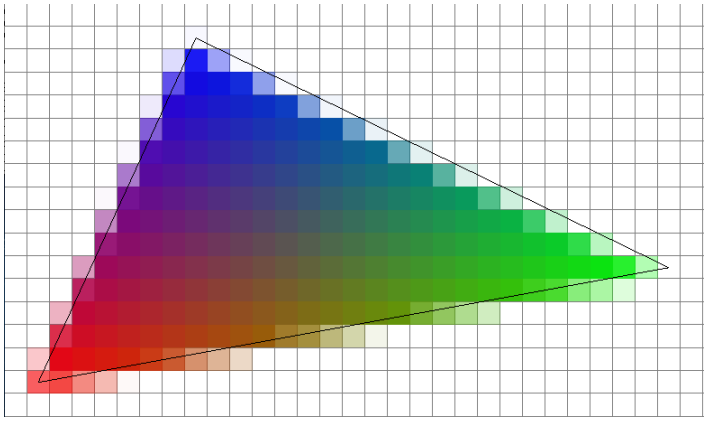

Figure 2.13: Scanline Converted Triangle
Source: http://web.eecs.umich.edu/~sugih/courses/eecs487/pa1.html

The *fragment shader* (alternatively referred to as *pixel shader*) is executed per fragment. The fragment shader computes and outputs a color value for a certain pixel in the frame (written to the framebuffer). The fragment shader is where illumination computation is usually performed.

Before a fragment actually becomes a pixel, it has to pass several tests – scissor test, stencil test and depth test – which could all be individually turned on or off, as well as configured. If the tests are passed, the fragment is written to the framebuffer.

This section went through the whole pipeline and gave a general picture of GPU workflow and dataflow. In the following section, the main function of a *renderer* in a graphics engine is explained.

2.2.4 Concept of a Renderer

Rendering is the process of generating an image for display from a scene representation. It is still the responsibility of the programmer to define how scenes are to be rendered. The GPU provides all the stages presented in Figure 5 and all the horsepower, but rendering does not happen automatically. Precisely defining the mathematical model and then making the CPU and GPU execute this model is the job of a graphics programmer.

For mainstream renderers, access to GPU hardware and implementation of the mathematical model is done using a *graphics application programming interface* (API) with a compatible *shading language*. The two most popular APIs are listed in Table 2.2. In a similar way that a conventional programming language (e.g. C++) with its standard library facilities exempt the programmer from writing assembly code and accessing raw memory, the graphics API along with the shading language provides higher-level access to GPU hardware.

Table 2.2: Graphics APIs

API Name	Platform	Shading Languages	Organization
OpenGL	Windows, Linux, OSX	GLSL, Cg	Khronos
Direct3D	Windows	HLSL, Cg	Microsoft

The software produced by a programmer to do rendering is called a *renderer*. Because a renderer has to run real-time with interactive frame-rates, straightforward implementation of a chosen mathematical model is usually not good enough. Graphics programmers look for tradeoffs and exploit hardware strengths to build the most efficient solutions. The same mathematical model may be implemented in various ways with different performance characteristics and a lot of research in the

field is concentrated on this problem – finding the fastest way to render content on current and future generations of hardware.

2.3 Forward Shading

Forward shading is the straightforward application of GPU pipeline to render an object. That is, triangles representing an object are submitted to the GPU, transformed, clipped and then rasterized; the resulting fragments are shaded and written to the framebuffer – all done in *one draw call*, i.e. one invocation of the GPU pipeline. During shading, contribution of one or several light sources is calculated.

There are some problems, inherent to this approach, which occur when trying to scale forward shading to many light sources. For instance, if only one light's contribution is computed in the fragment shader (*multi-pass* forward rendering), but the object is affected by many lights, then this object has to be re-rendered many times with different input to the fragment shader (different per-light data) and results should be additively blended into the framebuffer. In this case, a lot of unnecessary processing is done – same transformation is applied to vertices multiple times, same vertices are rasterized multiple times and bandwidth is wasted by writing to the framebuffer at the end of each pass, instead of computing the sum of all contributions and writing it once. The benefit of this approach is that it is the easiest forward shading implementation to program.

Alternatively, contribution of multiple light sources can be calculated in the shader, so as to apply all lighting to an object at once – in one invocation of the GPU pipeline (*single-pass*). If the list of lights is different for each object and there are thousands of objects, then light data has to be switched before rendering each object,

resulting in delays due to GPU-CPU synchronization and hindering performance severely.

The workaround is to batch objects by lights, i.e. form groups of objects affected by same lights and dispatch them to GPU without intermediate data uploads. Doing this requires certain compromises. Because perfect grouping would mean having too many groups (potentially as many, as there are objects), grouping has to be rather coarse. This implies possibly skipping lights for objects that should be affected by those lights, or instead doing computation for lights that in fact do not reach the objects rendered.

If these problems are ignored and perfect grouping is done with the assumption of instant light data upload without synchronization issues, there is still another big problem. With forward shading, even if a light source affects only a small part of an object, the whole object has to be transformed and rasterized, and each resulting fragment must be shaded. For example, suppose that there is a wall with a torch on it. The torch is significantly affecting only a small part of this wall. The wall covers the whole screen and the lit part occupies only a small part of the screen. In this case shading is actually necessary only for a small percent of fragments (those which receive light). Nevertheless, with forward shading, all fragments will be processed, resulting into a lot of useless work.

A possible workaround for this case is to break down big objects into smaller pieces. However, it trades-off geometry batching efficiency. Consequently, the right balance can never be reached. For this reason, having many small objects is not a suitable solution either.

The core of these problems is tight coupling of geometry operations (vertex processing) and lighting operations (fragment processing). Performing lighting in a

separate stage, after all geometric computation is done, is the idea behind *deferred shading* (shading is being *deferred*, hence the name) which is described below.

2.4 Deferred Shading

Deferred shading (Saito and Takahashi, 1990) at its core is separation of lighting calculations for each pixel on screen from obtaining per-pixel attributes necessary to do those lighting calculations. Classic deferred shading is done in two passes, namely *geometry pass* and *lighting pass*. In the *geometry pass*, all necessary information for each pixel is resolved and stored in the *G-buffer* (geometric buffer). In the lighting stage, the G-buffer is sampled and lighting calculations are performed (Hargreaves, 2004).

Figure 2.14: G-buffer Components in the Renderer of the Game Killzone 2 (Valient, 2007)
Left Top to Right Bottom: Depth, Normal, Specular Intensity, Specular Power, 2D Motion Vector, Diffuse Color

Figure 2.15: Final Image of the Game Killzone 2 (Valient, 2007)

During the geometry pass, all vertices are transformed, triangles are rasterized, and all necessary per-pixel information about visible geometry is stored into the G-buffer. The G-buffer is essentially a set of textures components of which store per-pixel geometry information such as pixel view space position, diffuse color, normal vector, specular parameters and potentially any other piece of data (Valient, 2007). Figure 2.14 illustrates the G-buffer components used in the engine for the game Killzone 2, developed by Guerilla Games. Figure 2.15 shows the final frame image produced using information from the G-buffer.

When the color of the actual pixels needs to be computed, i.e. when lighting or any post-processing is done (motion blur, screen-space ambient occlusion etc), information from the G-buffer is sampled. This way any lighting or post-processing becomes just an additional screen-space pass.

In order to develop a more clear understanding of the difference between forward and deferred shading, let us juxtapose them using their loosely formulated algorithms:

Forward shading:

```
for each light in Lights
    for each object in Objects
        transform
        shade
    end for
end for
```

Deferred shading:

```
for each object in Objects
    transform
    output per-pixel geometry data
end for
for each light in Lights
    shade
end for
```

The inner and outer loops in forward shading can be re-ordered, and the choice depends only on which one will be faster (depends on implementation).

From the pseudo-code given above, it can be seen that deferred shading separates the nested loops of the forward shading algorithm and executes them separately in a sequence. This characteristic is exactly what gives deferred shading its nice algorithmic property of shading load being independent from geometric load. A more detailed description of deferred shading and an in-depth look at its properties can be found in a survey done by Thaler (2010).

When deferred shading started being actively talked about in the game development industry from around 2004 and later, it may have seemed that developers came up with a new and original way of handling rendering in their engines. In fact, the approach itself is not new at all, and has been considered by researchers many years ago.

Deering et al (1988) was the first to introduce the idea of shading a pixel only once, after its visibility has been determined. At the time it was not called deferred shading and no G-buffer was used.

Modern version of deferred shading with a G-buffer was first proposed by Saito and Takahashi (1990). With this approach geometry processing is separated from shading (thus geometry is processed only once) and lighting or any visual enhancement is applied as a post-process using information from the G-buffer. Authors did not call it deferred shading either.

Ellsworth et al. (1991) is probably the first to use the name *deferred shading*. Authors used it on Pixel-Planes 5, which was a massively parallel SIMD machine, and noted that deferred shading avoids redundant calculations and conveniently maps to parallel architectures.

Creators of PixelFlow (another parallel machine for real-time rendering) incorporated deferred shading into their initial architecture (Molnar et al., 1992) and retained it as PixelFlow evolved into its final realization (Eyles et al. 1997).

As any good idea, deferred shading evolved over time into the modern approach we know today. Still, no real-time implementation on mainstream graphics cards or game consoles was attempted up until 2001.

The very first implementation of deferred shading in a commercial game was done by Geldreich in “Shrek” on Xbox1 in 2001. He and his colleagues later gave a

talk (Geldreich, 2004) at GDC2004 with demos on PC (DX9), Xbox1 and PlayStation2. Hargreaves also gave talks on deferred rendering at GDC2004 (Hargreaves, 2004). Since then more and more developers started looking into deferred shading and publishing their results.

As most mainstream hardware obtained *multiple render targets* (MRT) capabilities, and GPUs grew faster, deferred shading implementations started appearing in commercial games. For example, Shishkovtsov (2005) discusses implementation in "STALKER", going into detail on design decisions, tradeoffs and optimizations. Valient (2007) looks at implementation of deferred shading in "Killzone2" on Playstation3. Koonce (2008) discusses problems and solutions found during work on "Tabula Rasa". Filion (2008) elaborates on why deferred shading was chosen for "Starcraft2" and what special effects this allowed to achieve.

Main disadvantages of deferred shading, which result into significant performance penalties, are increased bandwidth and GPU memory storage. As screen resolution increases the renderer becomes bandwidth bound. These shortcomings of deferred shading motivated development of variations called *light pre-pass* (also referred to as *deferred lighting*) (Engel, 2009) and *inferred lighting* (Kircher, 2009).

Light pre-pass consists of three passes: geometry pass, light pass and forward pass. Geometry pass outputs normals and specular exponent into a single render target (RT). Light pass evaluates diffuse and specular equations, accumulating these values for each light in two RTs. Alternatively specular light accumulation color may be discarded and only specular luminosity written into alpha channel of diffuse RT, thus reducing RTs to one. Forward pass renders geometry the second time, fetching diffuse and specular material colors and applying them on accumulated diffuse and specular light values. The benefits of light pre-pass are possibility of using only one

RT (making it suitable for older and embedded hardware), reduction in bandwidth consumption on per-fragment attributes and also capability of utilizing materials. If geometric complexity of the scene is high, overhead of second pass on geometry may result in worse performance than deferred shading.

Inferred lighting is very similar to light pre pass and uses three passes as well: geometry pass, light pass and material pass. The first two render at lower than framebuffer resolution. Lighting information is upscaled in the material pass using a trick with a *discontinuity sensitive filter* (DSF), avoiding severe lighting artifacts. This allows significant savings in bandwidth. Inferred lighting also provides a solution for rendering translucent objects, in which alpha geometry is rendered with a stipple pattern in geometry pass. With DSF this allows reconstruction of both translucent and opaque geometry from the G-buffer.

Current state-of-the-art is *tiled deferred shading* first used by Engstad (2008) in Uncharted, popularized by Andersson (2009) in Battlefield 3 and promoted from research side by Lauritzen (2010, 2012). The idea is to build a screen-space grid, intersect all light sources with the grid and compose a light list for each tile. The lighting shader is then executed for each pixel only once, fetching light data for the relevant tile and applying all relevant lights in one pass. This way during shading the G-buffer is read only once per pixel alleviating the bandwidth overhead problem that plagues deferred shading.

A spin-off of this idea is *tiled forward shading* introduced by Olsson (2011), which handles transparency and anti-aliasing easily while keeping the benefits of many cheap lights.

Many more companies use deferred shading or deferred techniques, but do not publish their results. One could say that by now deferred shading has become

mainstream. However, there are few publications with detailed performance analysis of deferred shading versus forward shading. Obviously results will vary a lot, because they would heavily depend on various factors: target hardware, geometric complexity of scenes and their layout, how much computation does it take to shade a pixel in a certain engine etc. Unfortunately it is not possible to establish a universal benchmark, because each solution is developed with specific needs in mind, incorporating different features and different implementations of these features. Nevertheless it is still useful to provide detailed results from case studies, because even if "your mileage may vary", more data about technique's behaviour becomes available, which helps understand what the general performance trend is for this particular technique.

Chapter 3

EXPERIMENTAL SETUP

3.1 Renderer and Hardware Setup

In this study standard deferred shading renderer and multi-pass forward shading renderer are implemented. Below is a description of features shared by both renderers. Features relevant to individual implementations are described in the corresponding subsections.

Four types of light sources are supported, namely ambient, directional, point and spot lights. Directional, point and spot lights can be rendered with or without the use of shadow maps. In our implementation, we did not combine ambient and directional light passes, because it is assumed there could be more than one directional light.

Since both implementations compute lighting contribution for one light at a time, shadow map textures are reused. Directional lights and spot lights have different requirements for shadow map resolution, so a 512x512 depth texture is used for spot lights and a 2048x2948 one is used for directional lights. Point lights use a 512x512 depth cube map and rendering is done into each face separately (using layered rendering to do everything in one pass proved to be several times slower). Each light uses an appropriate light list (discussed below) for generating shadow maps.

In order to avoid redundant computation, *light lists* are built for each light. In other words, for each light, a list of objects affected by the light is composed by intersecting light bounding volumes with objects' axis-aligned bounding boxes (AABBs) by performing collision-detection tests. Also, a *camera visibility list* is

built – objects visible to the camera are determined by intersecting the camera bounding volume with objects' AABBs. Additionally, *visibility light lists*, which are set intersections of light lists and the camera visibility list, are built. How each renderer uses those lists is mentioned in respective sections. Some of the collision-detection code is based on (Ericson, 2004).

Light contribution computation is done per-fragment and uses normal mapping. For this reason per-vertex information includes texture coordinates, a normal and a tangent. Two textures are used per fragment, namely diffuse and normal map.

Both implementations write lighting computation results into a 16-bit floating point texture in order to perform high dynamic range (HDR) rendering (Debevec, 1997). This accumulation buffer is then used for post-processing, consisting of tone-mapping and blooming. Reinhard tone-mapping operator is used in this study (Reinhard, 2002). In the final pass, blooming, tone-mapping and light accumulation buffers are used to write the end result into the default framebuffer.

3.1.1 Collision Detection

The term *collision detection* is usually used in the context of physics calculations. However, it is useful not only for physics simulations. Collision detection is also generally used to avoid unnecessary computation. Specifically, if collision detection is not performed to determine visible objects, all objects in the scene have to be submitted to the pipeline and transformed only to discard many of them later on because they lie outside the view frustum. This is the reason why collision detection algorithms are used to perform intersection tests between objects' bounding volumes and camera view volume (to determine visible objects), as well as between objects' bounding volumes and light volumes (to determine objects affected by each light).

A widespread bounding volume type for objects is *axis-aligned bounding box* (AABB). An AABB of an object consists of a minimum and a maximum extent of an object along each axis. That is, for each of the three axes (x,y,z) a minimum and a maximum value are stored. These six numbers are enough to represent an AABB.

To determine the objects inside the view frustum, the view pyramid and objects' AABBs are tested for intersection. The same test is used for obtaining the light list for spot lights. Bounding pyramids are found for spot lights' cones and then tested against objects' AABBs.

To perform this *pyramid-AABB* test, the pyramid is broken down into five planes (base and four sides) such that planes' normals point inside the pyramid. Having three vertices on a plane, **v1**, **v2** and **v3**, specified in counter-clockwise order, the plane equation components **n** and *d* are obtained:

$$\mathbf{n} = \frac{(\mathbf{v2} - \mathbf{v1}) \times (\mathbf{v3} - \mathbf{v1})}{\|(\mathbf{v2} - \mathbf{v1}) \times (\mathbf{v3} - \mathbf{v1})\|} \tag{3.1}$$

$$d = \mathbf{v1} \cdot \mathbf{n} \tag{3.2}$$

where **n** is the plane unit normal and *d* is the closest signed distance from plane to origin. The × symbol signifies vector cross-product. Vector cross-product is defined such that in an expression $\mathbf{c} = \mathbf{a} \times \mathbf{b}$ magnitude of vector $\boldsymbol{c}$ is defined to be $||\mathbf{c}|| = sin(\theta) \cdot ||\mathbf{a}|| \cdot ||\mathbf{b}||$ (where θ is the angle between vectors **a** and **b**) and the unit vector $\hat{\mathbf{c}}$ is defined to be perpendicular to both **a** and **b** in the direction given by the right-hand rule.

The plane equation is used to determine which side of the plane (relative to the plane normal) a vertex lies. If distance from plane to vertex is positive, the vertex lies in the half-space the plane normal points into. Conversely, if the distance is negative,

the vertex lies in the half-space opposite the plane normal direction. Distance from AABB's points to a plane is calculated as:

$$\mathbf{v} \cdot \mathbf{n} - d \tag{3.3}$$

where **v** is the vertex tested.

To determine if a vertex lies inside the pyramid, signed distances from all five planes to the vertex have to be positive:

$$\mathbf{v} \cdot \mathbf{n1} - d1 > 0 \;||\; \ldots \;||\; \mathbf{v} \cdot \mathbf{n5} - d5 > 0 \tag{3.4}$$

If at least one point of AABB is inside the pyramid, the AABB intersects the pyramid and the pyramid-AABB test returns *true*.

For building the point light list, another type of test is necessary. In this case a sphere-AABB intersection test has to be performed. Distance from center of the sphere to the AABB is calculated. If distance is less than sphere radius, the sphere intersects the AABB. The pseudo code for the functions IntersectSphereAABB and DistSquareFromPointToAABB, which are used to perform the test described, are taken from (Ericson, 2004) and provided below.

```
IntersectSphereAABB(sphere, aabb)
begin
    return DistSquareFromPointToAABB(sphere, aabb) < sphere.center
end


DistSquareFromPointToAABB(point, aabb)
begin
    distance <= 0
    if (point.x > aabb.max.x)
```

```
        distance += (point.x - aabb.max.x) * (point.x - aabb.max.x)
    end if
    if (point.x < aabb.min.x)
        distance += (aabb.min.x - point.x) * (aabb.min.x - point.x)
    end if
    repeat the two if statements above for Y and Z coordinates
    return distance;
end
```

For the purposes of producing point light shadow maps for each face of the cube map, the bounding sphere of the point light is split into six pyramids (approximately spanning the bounding sphere). Before the objects are rendered into each face of the cube map, the objects from the point light list are also intersected against the above mentioned pyramids. This is done with the pyramid-AABB intersection test described previously.

3.1.2 Shading

Illumination for each fragment is computed according to the Phong reflection model (Phong, 1975). The RGB fragment color for the four types of light sources (ambient, directional, point and spot) is calculated using the following formulas:

$$RGB_{ambient} = RGB_D \times RGB_I \tag{3.5}$$

$$RGB_{directional} = (RGB_D \times D + RGB_S \times S) \times RGB_I \tag{3.6}$$

$$RGB_{point} = (RGB_D \times D + RGB_S \times S) \times RGB_I \times A \tag{3.7}$$

$$RGB_{spot} = (RGB_D \times D + RGB_S \times S) \times RGB_I \times A \times F \tag{3.8}$$

where RGB_D is diffuse color of the surface material, RGB_S is specular color of the surface material, RGB_I is intensity of light, D is the diffuse term, S is the specular term, A is light attenuation due to distance from light source and F is the spot light factor.

The diffuse term D is calculated as:

$$D = \mathbf{N} \cdot (-\mathbf{L}) \tag{3.9}$$

where **N** is fragment normal and **L** is light incidence vector. D is clamped to the [0…1] range.

The specular term S is calculated as:

$$S = (\mathbf{R} \cdot \mathbf{C})^p \tag{3.10}$$

where **R** is light incidence vector (**L**) reflected relative to fragment normal (**N**), **C** is camera direction vector and p is specular power of the surface material The dot product is clamped to [0…1] range before the exponent is applied.

Light attenuation term A is calculated as:

$$A = \frac{1}{1+d^2} \tag{3.11}$$

where d is the distance to the light source.

The spot light factor F is calculated as:

$$F = \frac{O_{cos} - S_{cos}}{O_{cos} - I_{cos}} \tag{3.12}$$

where O_{cos} is the outer angle cosine, I_{cos} is the inner angle cosine and S_{cos} is the cosine of the angle between light direction and light incidence vectors.

When shadows are enabled, directional, point and spot light color is also multiplied by a shadow factor. The shadow factor is in range [0, 1] and represents obstruction of the object surface from the light source by other objects. If the fragment is in shadow, shadow factor is equal to 0. In this case fragment color, after being multiplied by zero, becomes black, i.e. not illuminated. Conversely, if the fragment is not in shadow, shadow factor is equal to 1. Consequently, the fragment color is multiplied by one, preserving the computed color.

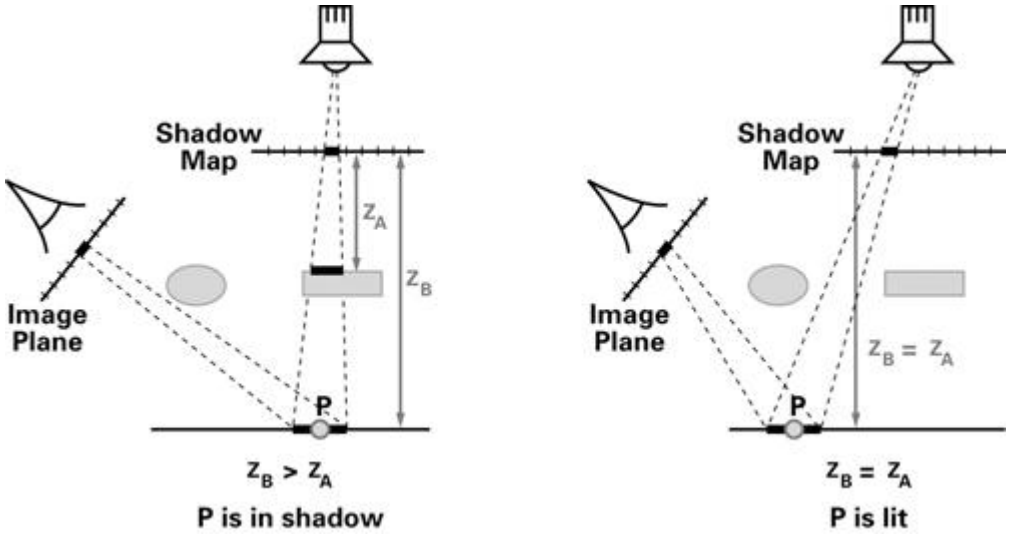

Figure 3.1: Shadow Mapping
Source: http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html

Figure 3.1 illustrates the principle of shadow mapping. The scene is rendered from light's point of view with the resulting depth stored in the shadow map. During shading, the fragment's position is transformed into the coordinate space of the shadow map. The fragment's xy components are used as texture coordinates to sample the shadow map. The value fetched (Z_a) corresponds to the surface being lit

by the light. Z_a is compared to the fragment's actual depth (Z_b). If $Z_b < Z_a$, the fragment is in shadow, and if $Z_b = Z_a$, the fragment is lit.

In our implementation of shadow-mapping, hardware PCF (percentage-closer filtering) is enabled for the depth texture (Reeves, 1987). In the lighting shader, view space position of the fragment is multiplied by a matrix which applies the following chain of transformations: view space, world space, light space, clip space from light's perspective and window space (except x and y components are in [0…1] range). The resulting four-component vector is used for lookup from the shadow map texture.

3.1.3 Forward Renderer

The forward renderer implementation uses the *multi-pass* approach for light accumulation. Each light requires a separate pass and only one light's contribution is computed at a time in the fragment shader.

Camera visibility list is used for ambient light and directional lights shading since both light types affect all objects in the scene and only objects seen by the camera should be shaded. Point lights and spot lights use visibility light lists to render and shade appropriate objects, because only the objects that are visible to the camera and are affected by the light need to be rendered.

On a high level, the rendering loop consists of two steps:

Rendering loop

```
begin
    ShadingPass()
    Postprocessing()
end
```

The shading pass can be described with the following pseudo code:

Shading pass:

```
begin
    AmbientLightPass(ambient_light)
    for each directional_light in DirectionalLights
        DirectionalLightPass(directional_light)
    end for
    for each point_light in PointLights
        PointLightPass(point_light)
    end for
    for each spot_light in SpotLights
        SpotLightPass(spot_light)
    end for
end
```

The four blocks of pseudo code below describe the four light passes used in the shading pass above.

Ambient light pass:

```
begin
    upload ambient light data
    for each object in VisibilityList
        upload transformation data
        draw object
    end for
end
```

Directional light pass:

```
begin
    if shadow enabled
        render directional light shadow map
    end if
    upload directional light data
    for each object in VisibilityList
        upload transformation data
        draw object
    end for
end
```

Point light pass:

```
begin
    if shadow enabled
        render point light shadow map
    end if
    upload point light data
    for each object in LightList
        upload transformation data
        draw object
    end for
end
```

Spot light pass:

```
begin
    if shadow enabled
        render spot light shadow map
    end if
    upload spot light data
    for each object in LightList
        upload transformation data
        draw object
    end for
end
```

3.1.4 Deferred Renderer

Deferred renderer implementation uses the standard approach with the geometry pass and the shading pass (no light pre-pass or inferred lighting used). G-buffer consists of *normal* information, *diffuse* color and position information in the form of *fragment depth*, as illustrated in Figure 3.2.

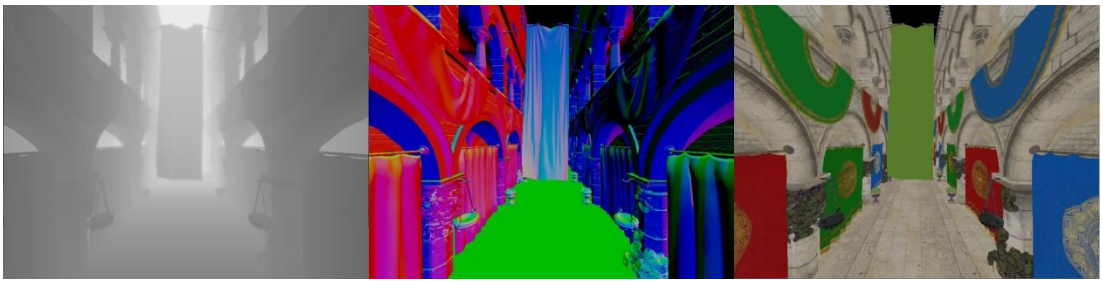

Figure 3.2: G-buffer Render Targets
Left to Right: Depth, Normals, Diffuse Color

View space position is reconstructed using screen space fragment coordinates (window position and depth). For both normal and diffuse buffers, RGB8 render targets are used (8 bits per color component). Depth is calculated as part of geometry pass rasterization, which helps avoid additional overhead for storing position data. Depth is stored in an FP32 depth texture (32-bit floating point). This constitutes a rather slim G-buffer – only 10 bytes need to be read per fragment. Tests with "fatter" formats for normal and diffuse data showed that bandwidth problems crippled performance.

In the geometry pass, camera visibility list is used to rasterize all visible geometry and dump geometry information into the G-buffer. During shading, crude light volume geometry is used for point and spot lights (low poly spheres and cones), and full-screen quads are used for ambient and directional lights.

For view space fragment position reconstruction, the following procedure is performed. Window space x and y components are divided by respective window resolution values. The z value is sampled from the depth texture. This xyz triple is used as the position to be multiplied by a matrix. The matrix is the inverse of the following chain of transformations: view space, clip space and window space (except x and y components being in [0…1] range). After multiplying the above mentioned position by the matrix, the resulting four-component vector is divided by the fourth

component. Post-division, the first three components of the vector are the view space fragment position.

The rendering loop of the deferred shading renderer consists of three steps:

Rendering loop:

```
begin
    GeometryPass()
    ShadingPass()
    Postprocessing()
end
```

The geometry pass can be described with the following pseudocode:

Geometry pass:

```
begin
    for each object in VisibilityList
        upload transformation data
        draw object (output to Render Targets)
    end for
end
```

The shading pass is identical to the one in the forward shading renderer, however each light pass (inside the shading pass) is different.

Ambient light pass:

```
begin
    upload ambient light data
    draw fullscreen quad
end
```

Directional light pass:

```
begin
    if shadow enabled
        render directional light shadow map
    end if
    upload directional light data
    draw fullscreen quad
```

```
end
```

Point light pass:

```
begin
    if shadow enabled
        render point light shadow map
    end if
    upload transformation data
    upload point light data
    draw unit sphere
end
```

Spot light pass:

```
begin
    if shadow enabled
        render spot light shadow map
    end if
    upload transformation data
    upload spot light data
    draw unit cone
end
```

3.1.5 Hardware and System Software

Tests were done on a laptop with 2.1 GHz Dual-Core AMD Athlon II, 4GB RAM and AMD HD 5145 video card with 512 MB VRAM, running Windows 7 operating system. Catalyst 13.1 driver was used for the video card (latest legacy driver available). All tests were run under 1366x768 resolution (highest available) and one particular test was done under 800x600.

3.2 Scene Setup

For the experiments to be representative of actual performance “in real world”, a sufficiently complicated scene should be used. For example, if there is not enough

geometric complexity, vertex transformation becomes very cheap and results are skewed. Similarly, if there is not enough object overlap, the overhead of overdraw is underplayed.

In our experiments, the widely used Atrium Sponza Palace model from (Meinl, 2010) was used. It is a good approximation of a game scene for purposes of our work and the result image can be seen in Figure 3.3.

Figure 3.3: The Atrium Sponza Palace Scene Shaded

The Sponza model is broken down into sub meshes. This allows to treat them as separate objects and to do culling. Performing culling is very important to get a good approximation of how a renderer with deferred or forward architecture would behave.

Constructing different illumination layouts for up to a hundred of light sources by hand is impractical. For this reason, positions, intensities and other light parameters are generated randomly. In order to get desirable characteristics within some limits

but still observe variation, appropriate bounds are set for random distributions and generated numbers are then adjusted to meet needed constraints. For tests to be repeatable, a fixed seed value is used for the random number generation engine.

3.3 Tests Description

It is expected that deferred and forward rendering approaches have different performance characteristics under different conditions. We wish to explore the following illumination conditions:

- big lights (large influence area) have shadows enabled (Figure 3.6)
- big lights without shadows (Figure 3.5)
- small lights without shadows (Figure 3.4)
- mixed case with many small lights and a few big shadow-casting lights

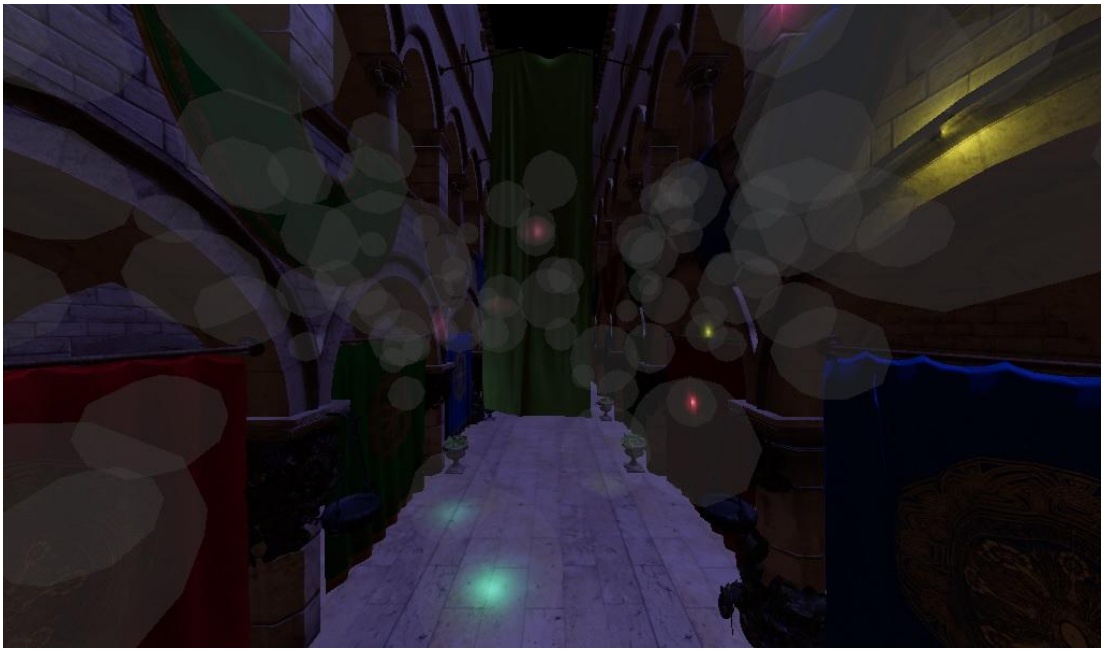

Figure 3.4: Sponza Scene with Many Small Lights (Bounding Volumes Are Drawn)

Figure 3.5: Sponza Scene with Many Non-Shadow Casting Big Lights

Figure 3.6: Sponza Scene with Many Shadow-Casting Big Lights

For each case the number of light sources is varied to see the general behavior of the rendering technique tested. Number of lights is scaled from 1 to 100, and tests are

taken in increments of one light for [1-10] range, in increments of two for (10-20] range and increments of ten for (20-100] range.

At all times one ambient light and one directional light are present. When shadows are enabled for big light sources, directional light also uses shadows.

The performance data collected is *average frame time* and *average fps* (frames per second) for a 20 second run. Frame time is used to compare speed of shading methods and fps is used to get a general picture of how observed performance difference translates into perceived user experience difference.

Chapter 4

EXPERIMENTAL RESULTS

In order to evaluate performance of deferred and forward renderers with variable number of lights, two metrics, specifically, average time spent rendering a frame (frame time) and frames per second (fps) are employed. Frame time measures the cost of rendering, and it is generally used for comparing efficiency of rendering techniques. FPS is a user-experience metric. It helps to measure whether interactive frame rate (around 15 fps) is maintained at particular loads and it illustrates how fast user-experience degrades as the computation load increases.

4.1 Rendering Big Lights

In Figure 4.1, it can be seen that deferred variant exhibits smaller render time. In contradiction with the expectation that deferred would be slower for few lights and would catch up with forward as the number of lights increases, deferred variant is immediately faster. As seen from the figure, additional cost of maintaining the G-buffer is amortized at once, showing that deferred approach is suitable performance-wise even for small numbers of lights.

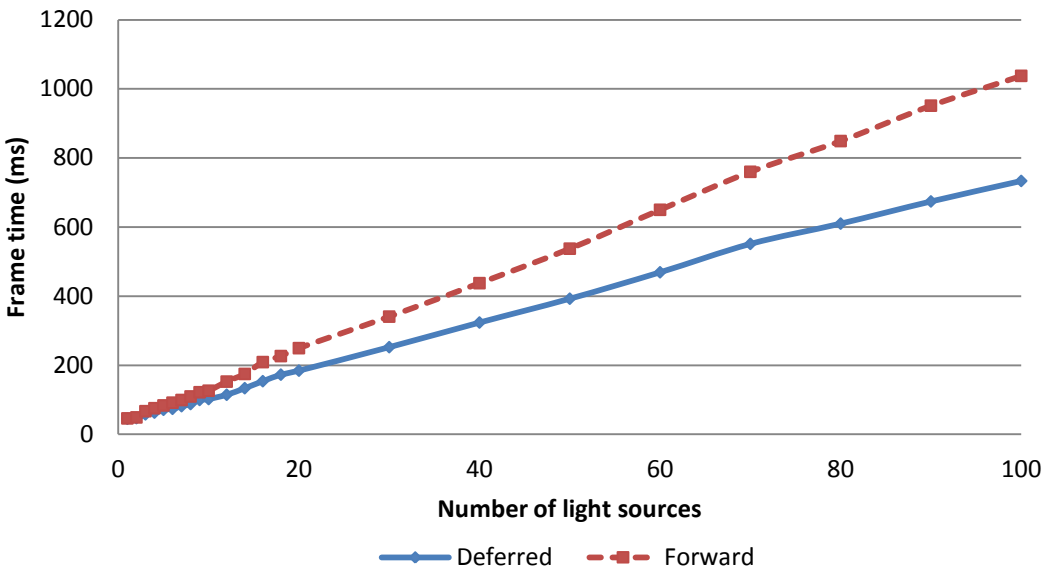

Figure 4.1: Frame Time While Rendering *Big Lights with Shadows*

It can be seen in the figure that for both deferred and forward approaches, the render time is linearly dependent on the number of lights where the cost for each additional light is almost constant, being roughly 7 ms for deferred and 10 ms for forward shading.

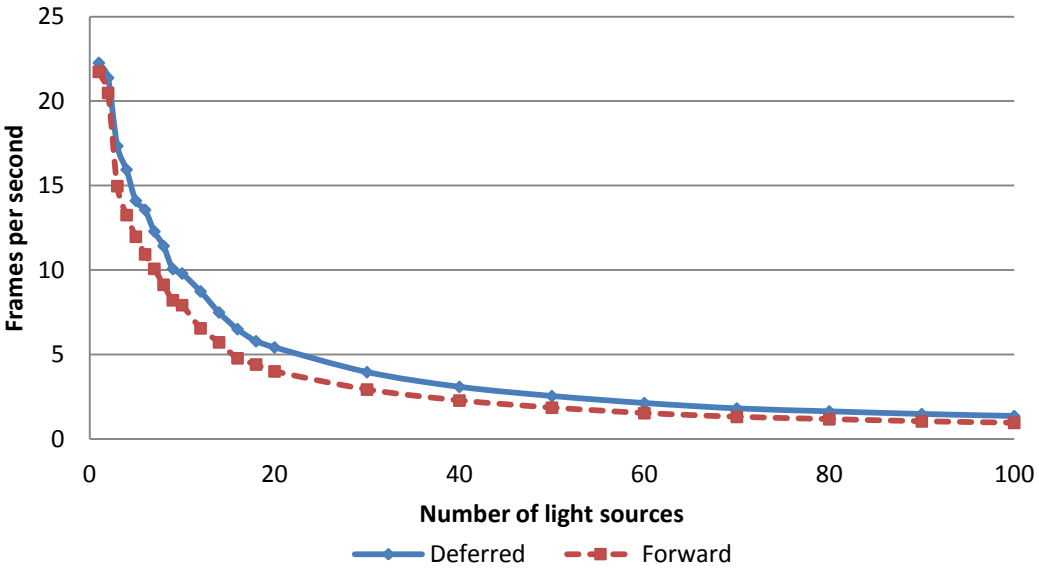

Figure 4.2: FPS While Rendering *Big Lights with Shadows*

Figure 4.2 presents the user-experience as light count increases. It can be seen that fps does not decrease linearly but exponentially. For the used hardware configuration fps drops below interactive (< 10-15 fps) very fast – to 8 fps for forward and 10 fps for deferred at only 10 lights. Although deferred shading is faster, the difference is not significant. As expected, rendering shadow maps takes the biggest fraction of the cost to render a light.

Figures 4.3 and 4.4 present results for big lights without shadow maps. Performance gap has widened and deferred shading is on average almost two times faster. Cost per light is now different – 4.6 ms in case of forward shading and 2.3 ms in case of deferred shading.

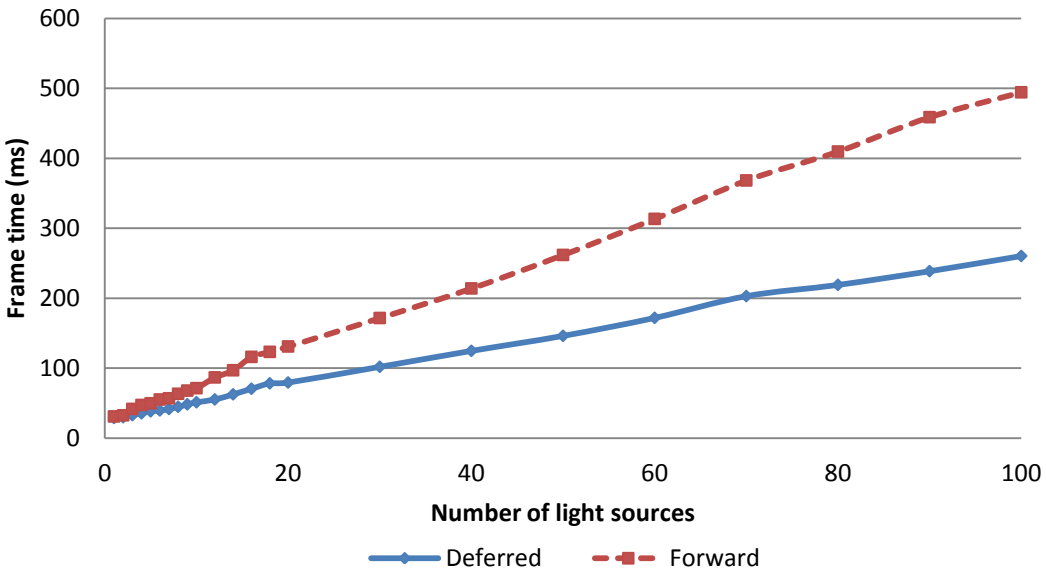

Figure 4.3: Frame Time While Rendering *Big Lights without Shadows*

As it can be seen in Figure 4.4, user-experience levels are very different from the previous case with enabled shadows. Forward shading drops to 10 fps at 14 lights,

whereas deferred degrades to 10 fps only at 30 lights. This test clearly shows superiority of deferred over forward shading.

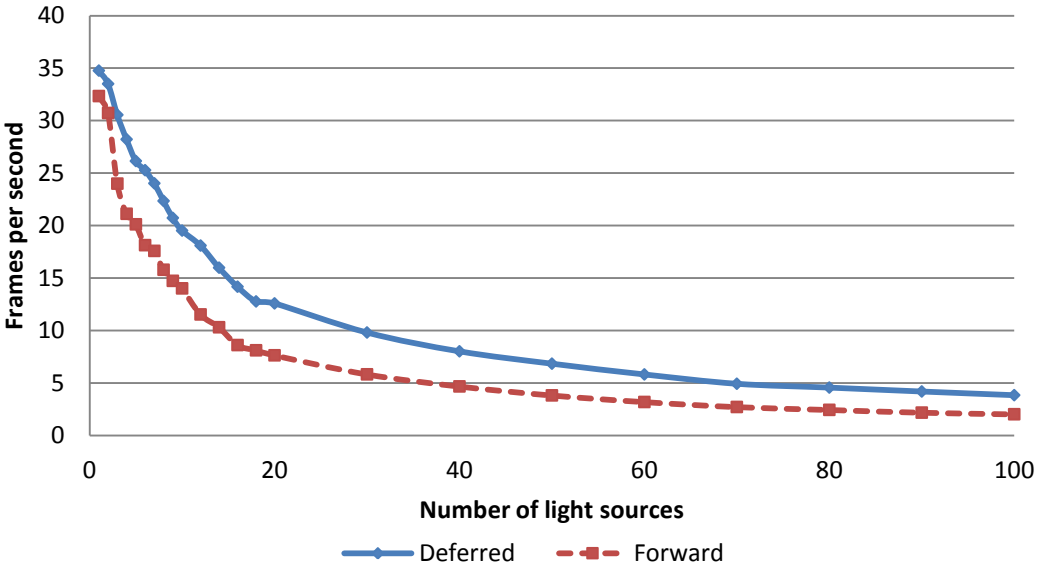

Figure 4.4: FPS While Rendering *Big Lights without Shadows*

Figures 4.5 and 4.6 illustrate the experimental results for a setup identical to the one above, except with lower resolution – 800x600 instead of 1366x768. Lower resolution means a smaller G-buffer and thus lower consumption of memory and bandwidth. Memory occupancy is not a significant factor, but decreased bandwidth usage is expected to result into higher performance for deferred approach.

It can be seen in Figure 4.5 that the gap between deferred and forward shading is wider than before. In Figure 4.6, after about 20 lights, deferred shading becomes 2 times faster and, as more lights are added, the difference becomes even higher. This confirms that deferred rendering is a bandwidth-hungry approach, and implies that at much higher resolutions performance of deferred will be impaired due to the need of sampling the G-buffer for many more pixels.

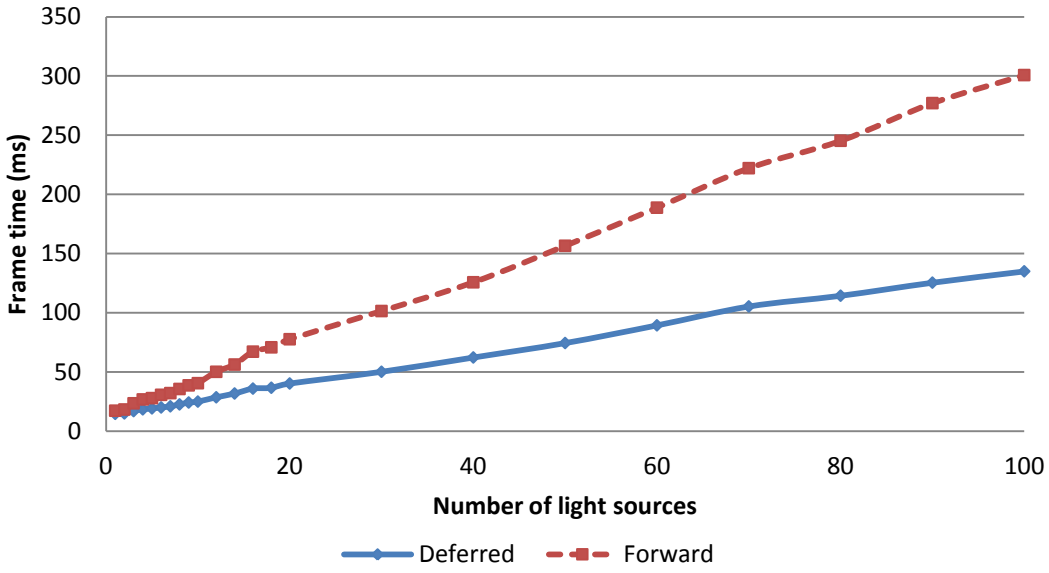

Figure 4.5: Frame Time While Rendering *Big Lights without Shadows* at 800x600 Resolution

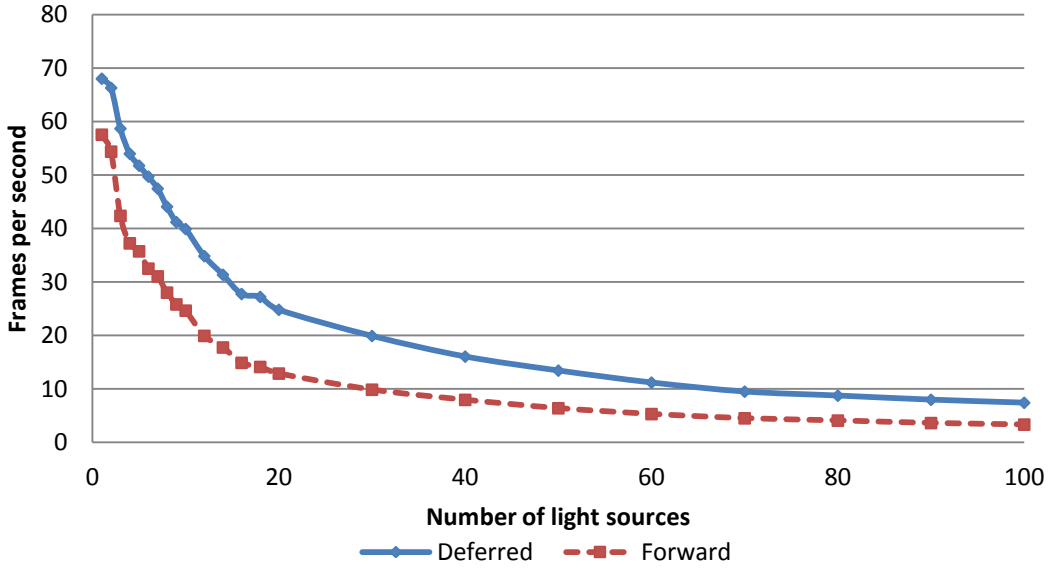

Figure 4.6: FPS While Rendering *Big Lights without Shadows* at 800x600 Resolution

4.2 Rendering Small Lights

Previous measurements were done for big lights covering from 20% to 100% of screen. Figures 4.7 and 4.8 present the results for small lights that cover 0.25% to at most 5% of the screen.

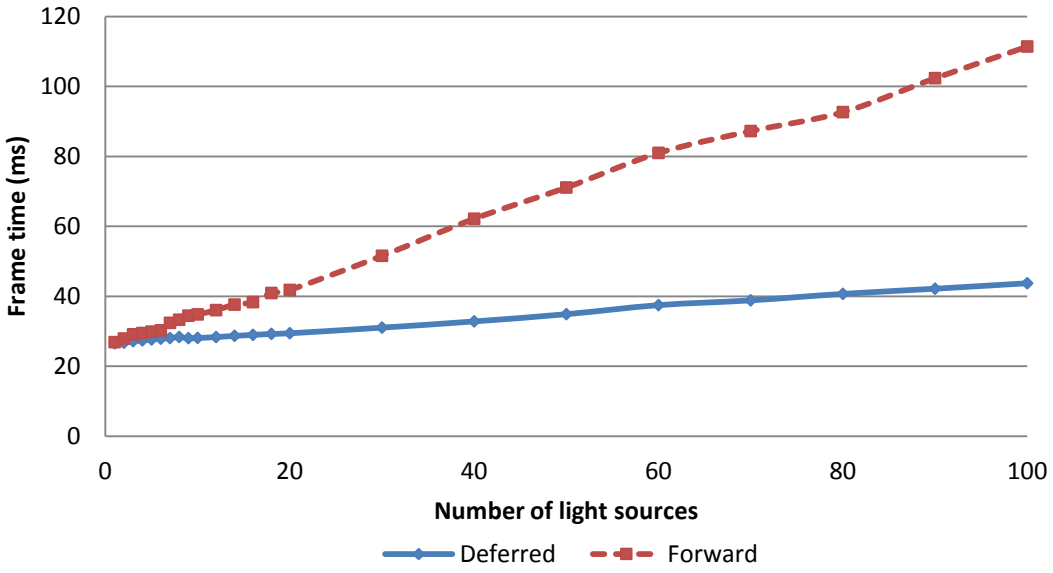

Figure 4.7: Frame Time While Rendering *Small Lights without Shadows*

It can be seen in Figure 4.7 that rendering small lights is a strong advantage of deferred rendering – rendering cost per light is 0.17 ms for deferred and 0.84 ms for forward. Deferred variant renders one small light 5 times faster than forward, resulting in very smooth fps degradation as number of lights grows. A linear dependence can be seen in Figure 4.8 which is a much more desirable behavior than exponential decay in previous tests.

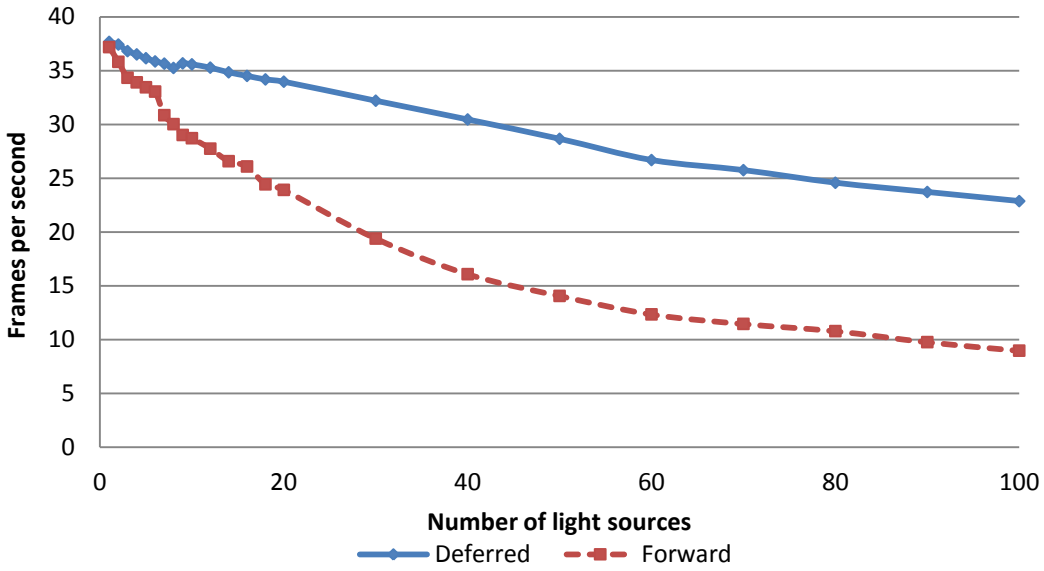

Figure 4.8: FPS While Rendering *Small Lights without Shadows*

The main reason for deferred shading exhibiting superior performance when rendering many small lights compared to forward shading is that the computational complexity is linearly dependent on the number of pixels affected by the light in the case of deferred shading whereas, for forward shading, it is linearly dependent on the screen space size of objects affected by the light. This means that, when a light affects only 10% of a pillar, deferred shading will compute illumination only for that 10% of the pillar. However, forward shading will do lighting calculations for the whole pillar, even though 90% of it is not affected by the light at all.

4.3 Rendering Big and Small Lights

The next four figures (4.9-4.12) reflect a more "real-world" situation. A typical scene in a modern game usually contains many small light sources which need not cast any shadows and only a few big light sources that cast shadows. From the previous tests it was seen that, with a deferred renderer, each additional big light with shadows comes at a cost of 7 ms and each additional small light takes 0.17 ms. With

the forward renderer, a big shadowed light costs 10 ms and a small one costs 0.84 ms. Thus, even if there are many small lights, each additional big light will bring performance of deferred and forward closer, because rendering time of big lights will dominate total frame time and the cost difference of deferred versus forward for big lights is not remarkably different.

To test this hypothesis, a ratio of "number of big lights to number of small lights" is picked and the tests are made for multiples of this ratio. For example, in Figures 4.9 and 4.10, the first data point reflects a scene with 1 big and 10 small lights, and the last data point – a scene with 10 big lights and 100 small ones. Figures 4.11 and 4.12 follow the same idea but with a different ratio – 1 to 5 instead of 1 to 10.

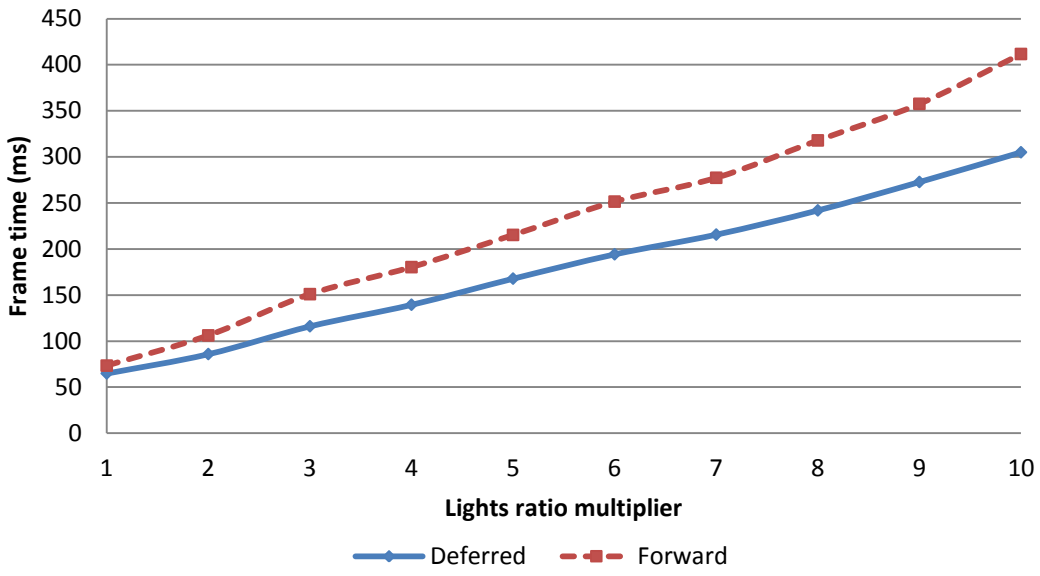

Figure 4.9: Frame Time While Rendering *Big and Small Lights* (1x10 Ratio)

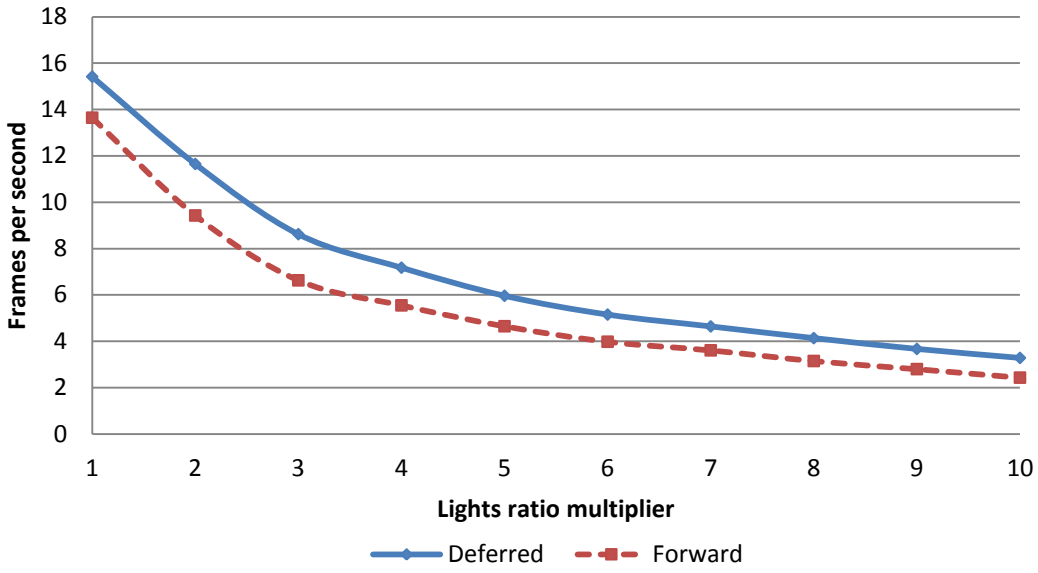

Figure 4.10: FPS While Rendering *Big and Small Lights* (1x10 Ratio)

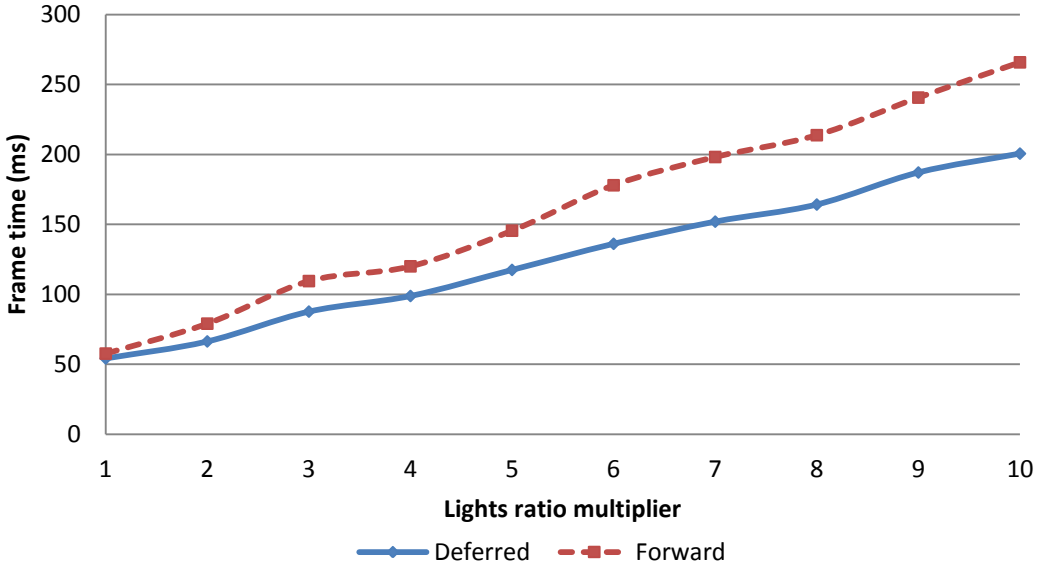

Figure 4.11: Frame Time While Rendering *Big and Small Lights* (1x5 Ratio)

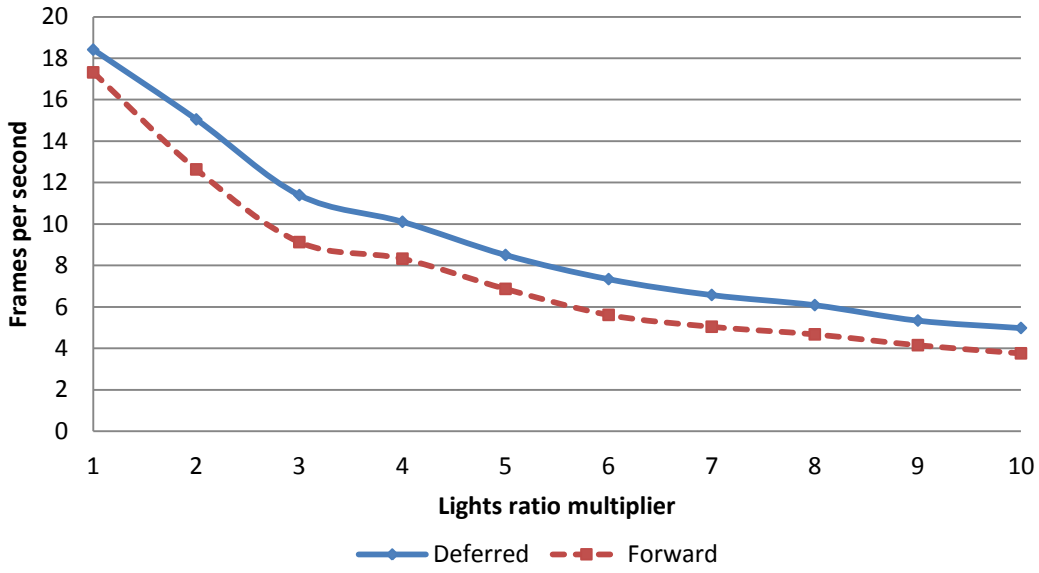

Figure 4.12: FPS While Rendering B*ig and Small Lights* (1x5 Ratio)

While there are minor differences between 1x10 and 1x5 ratio setups, the trend is very similar. Rendering big lights with shadows takes the majority of total frame time and thus, while still being faster, deferred shading does not show an advantage as significant, as was seen for rendering small lights exclusively.

Chapter 5

CONCLUSIONS AND FUTURE WORK

Two rendering approaches were benchmarked in this work, namely multi-pass forward shading and traditional deferred shading. The goal was to indentify how deferred and forward shading perform relative to each other in the following scenarios:

- many small lights
- many big lights
- many big lights with shadows
- many small lights and several shadow-casting big lights

Deferred shading was faster in every test for any number of lights. As the number of lights increased, deferred shading provided a bigger advantage in performance. As expected, the gap in performance between forward and deferred depended on the lighting setup.

For *small lights*, deferred shading provided the highest gain in performance over forward shading. In this scenario, per-light cost for deferred shading was very small and adding more light sources affected performance insignificantly, allowing high frame rate even with a hundred of lights.

With *big lights*, the advantage lessened, but was still very large, roughly two times faster for a high number of lights. However, when big lights *used shadows*, although visible and substantial, the difference was not as dramatic as in previous cases.

In the scenario with *many small lights* and *several big shadow-casting lights*, the picture was almost the same as for *big lights with shadows* exclusively. Shadow map rendering dominated the cost in the whole lighting computation time budget.

In summary, deferred shading was faster in every case and offered enormous benefits when rendering small light sources.

We did not test some more advanced current approaches – *tiled deferred* and *tiled forward rendering*. These are expected to perform significantly faster than traditional techniques and will be tested in our future work.

Another important item that didn't make it into experiments is skinned meshes (animated characters), which would put a lot more pressure on vertex transformation stage and presumably skew results in favor of deferred shading even more.

Also, results presented in this work were done on five year old mobile platform hardware. It would be useful to run the same tests on current desktop-class hardware and see if same behavior is observed.

REFERENCES

Andersson, J. (2009). Parallel Graphics in Frostbite – Current & Future. ACM SIGGRAPH 09, available online at: http://s09.idav.ucdavis.edu/ talks/04-JAndersson-ParallelFrostbite-Siggraph09.pdf

Debevec, P. (1997). Recovering high dynamic range radiance maps from photographs. SIGGRAPH 97, available online at www.pauldebevec.com/Research/HDR/debevec-siggraph97.pdf

Deering, M., Winner, S., Schediwy, B., Duffy, C., Hunt, N. (1988). The triangle processor and normal vector shader: a VLSI system for high performance graphics. ACM SIGGRAPH Computer Graphics, Vol. 22 (4), Pages 21-30.

Ellsworth, D., Tebbs, B., Neumann, U., Eyles, J., Turk, G. (1991). Parallel architectures and algorithms for real-time synthesis of high quality images using deferred shading. Workshop on Algorithms and Parallel VLSI Architectures, available online at: http://www.cs.unc.edu/techreports/92-034.pdf

Engel, W. (2009). Designing a renderer for multiple lights - the Light Pre-Pass renderer. ShaderX7.

Engstad, P. (2008). The Technology of Uncharted: Drake's Fortune. GDC, available online at: http://www.naughtydog.com/docs/Naughty-Dog-GDC08-UNCHARTED-Tech.pdf

Ericson, C. (2004). Real-time collision detection. Morgan Kaufmann.

Eyles, J., Molnar, S., Poulton, J., Greer, T., Lastra, A., England, N., Westover, L. (1997). PixelFlow: The Realization. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, Pages 57-68 , available online: http://www.cs.unc.edu/~jp/PxFl-hwws97.pdf

Filion, D., McNaughton, R. (2008). Starcraft II Effects and Techniques, Advances in Real-Time Rendering in 3D Graphics and Games. SIGGRAPH, available online: http://pds17.egloos.com/pds/200908/12/03/Chapter05-Filion-StarCraftII.pdf

Geldreich, R., Pritchard, M., Brooks, J. (2004). Deferred Lighting and Shading. GDC, available online at: http://www.tenacioussoftware.com/gdc_2004_deferred_shading.ppt

Hargreaves, S., Harris, M. (2004). 6800 Leagues under the sea. GDC, available online at: http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf

Hoef, M., Zalmstra, B. (2010). Comparison of multiple rendering techniques. Ultrecht University, available online at http://files.bassie-entertainment.com/Papers/Comparison-of-multiple-rendering-techniques.pdf

Kircher, S., Lawrance, A. (2009). Inferred Lighting: fast dynamic lighting and shadows for opaque and translucent objects. ACM SIGGRAPH 09.

Koonce, R. (2008). Deferred Shading in Tabula Rasa. GPU Gems 3, Addison-Wesley, available online at: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch19.html

Lauritzen, A. (2010). Deferred rendering for current and future rendering pipelines. ACM SIGGRAPH 10, available online at: http://download-software.intel.com/sites/default/files/m/8/1/1/9/3/40366-lauritzen_deferred_shading_siggraph_2010.pdf

Lauritzen, A. (2012). Intersecting lights with pixels: reasoning about forward and deferred rendering. ACM SIGGRAPH 12, available online at: http://bps12.idav.ucdavis.edu/talks/03_lauritzenIntersectingLights_bps2012.pdf

Meinl, F. (2010). The Atrium Sponza Palace, Dubrovnik. available online at: http://www.crytek.com/cryengine/cryengine3/downloads

Molnar, S., Eyles, J., Poulton, J. (1992). PixelFlow: high-speed rendering using image composition. ACM SIGGRAPH Computer Graphics Vol. 26 (2), Pages 231-240, available online at: http://www.cs.unc.edu/~molnar/Papers/pxfl.pdf

Olsson, O. (2011). Tiled Shading. JGT, available online at: http://www.cse.chalmers.se/~olaolss/papers/tiled_shading_preprint.pdf

Phong, B. T. (1975). Illumination for computer generated pictures. Communications of ACM 18 (1975), no. 6, 311–317.

Postma, B., & Isenberg, T. (2009). An evaluation of deferred shading under changing conditions. Order, 501, 2349, available online at: http://tobias.isenberg.cc/personal/papers_students/Postma_2009_EDS.pdf

Reeves, W., Salesin, D., Cook, R. (1987). Rendering antialiased shadows with depth maps. ACM SIGGRAPH Computer Graphics (Vol. 21, No. 4, pp. 283-291), available online at: http://graphics.pixar.com/library/ShadowMaps/paper.pdf

Reinhard, E. et al. (2002). Photographic tone reproduction for digital images. ACM, available online at http://www.cs.utah.edu/~reinhard/cdrom/tonemap.pdf

Saito, T., Takahashi, T. (1990). Comprehensible rendering of 3-D shapes. ACM SIGGRAPH Computer Graphics Vol.24 (4), Pages 197-206, available online at: http://www.cs.otago.ac.nz/cosc455/p197-saito.pdf

Shishkovtsov, O. (2005). Deferred Shading in S.T.A.L.K.E.R. GPU Gems 2, Addison-Wesley, available online at http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html

Thaler, J. (2010). Deferred Rendering. Vienna University of Technology, available online at http://www.cg.tuwien.ac.at/courses/Seminar/WS2010/deferred_shading.pdf

Valient, M. (2007). Deferred Rendering in Killzone. Develop Conference, available online at:http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf

Vince, J. (2006). Mathematics for computer graphics. Springer.