DEFERRED SHADING OF TRANSPARENT SURFACES WITH SHADOWS AND REFRACTION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ALİ DENİZ ALADAĞLI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN THE DEPARTMENT OF
MODELLING AND SIMULATION

MARCH 2015

Approval of the thesis:

DEFERRED SHADING OF TRANSPARENT SURFACES WITH SHADOWS AND REFRACTION

submitted by **ALİ DENİZ ALADAĞLI** in partial fulfillment of the requirements for the degree of **Master of Science in Game Technologies, Middle East Technical University** by,

Prof. Dr. Nazife Baykal _____
Director, **Informatics Institute**

Assist. Prof. Dr. Hüseyin Hacıhabiboğlu _____
Head of Department, **Modelling and Simulation**

Prof. Dr. Veysi İşler _____
Supervisor, **Computer Engineering**

Examining Committee Members:

Assoc. Prof. Dr. Alptekin Temizel
Modelling and Simulation, METU _____

Prof. Dr. Veysi İşler
Computer Engineering, METU _____

Assist. Prof. Dr. Hüseyin Hacıhabiboğlu
Modelling and Simulation, METU _____

Dr. Aydın Okutanoğlu
Simsoft, Ankara _____

Dr. Erdal Yılmaz
Argedor, Ankara _____

Date: 3 March 2015

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ALİ DENİZ ALADAĞLI

Signature :

ABSTRACT

DEFERRED SHADING OF TRANSPARENT SURFACES WITH SHADOWS AND REFRACTION

ALADAĞLI, ALİ DENİZ

M.Sc., Department of Modelling and Simulation

Supervisor : Prof. Dr. Veysi İşler

March 2015, 39 pages

Deferred rendering techniques became more widespread since graphic cards started becoming powerful enough to utilize such techniques effectively. While using additional lights with deferred rendering is more efficient than using them with forward rendering, deferred rendering has certain drawbacks, such as high memory usage and its inability to deal with transparency. However, transparent surfaces like glass are all around the real world and as such should be available in rendering engines that seek to simulate the real world. In this study a previously proposed method is used where a single layer of transparency is possible with deferred shading without resorting to a secondary rendering pipeline. This method is extended, as a part of the study, to incorporate refraction and shadows of transparent surfaces. The objective of these improvements is to partially overcome one of the major drawbacks of deferred rendering, that is transparency, by improving the visual fidelity of scenes containing transparency that use deferred shading while still adhering to the limitations of a real-time rendering engine. Performance analyses and comparisons are performed with other rendering pipelines where refraction and shadows of transparent surfaces are supported. The cost of simulating refraction in the fragment shader was found to be 4 extra texture instructions and 13 extra arithmetic instructions,

which manifested as a 10% decrease in FPS in a test scene with 60000 polygons. The cost of transparent shadows was found to be 13% at 2 lights and 34% at 64 lights with the test rig. When more than 4 lights were used the proposed method was at least 6% faster and at most 28% faster than a forward renderer with the same effects at 8 and 32 shadow casting lights respectively.

Keywords: Deferred,Transparency,Shadow,Refraction

ÖZ

SAYDAM YÜZEYLERİN GÖLGELER VE IŞIK KIRILMASI İLE GECİKMELİ GÖLGELENDİRİLMESİ

ALADAĞLI, ALİ DENİZ

Yüksek Lisans, Modelleme ve Simülasyon Bölümü

Tez Yöneticisi : Prof. Dr. Veysi İşler

Mart 2015 , 39 sayfa

Gecikmeli aydınlatma teknikleri, grafik kartları bu tekniklerden etkin bir şekilde faydalanabilecek kadar güçlenmeye başladıkça yaygınlaşmıştır. İlave ışıkları gecikmeli ışıklandırma ile kullanmak, aynı ışıkları ileri ışıklandırma ile kullanmaktan daha verimli olmasına rağmen, gecikmeli görsellemenin yüksek bellek kullanımı ve saydam yüzeylerin işlenememesi gibi belirli engelleri vardır. Halbuki, gerçek dünyada cam gibi saydam yüzeyler heryerde bulunabilir ve dolayısıyla gerçek dünyayı taklit etmeyi amaçlayan render motorlarında bulunabilmesi gerekmektedir. Bu çalışmada, tek bir saydamlık katmanının ikinci bir aydınlatma hattı kullanılmadan mümkün olduğu daha önceden öne sürülen bir yöntem kullanıldı. Bu yöntem, çalışmanın bir parçası olarak, saydam yüzeylerin gölgelerini ve ışığı kırmalarını kapsayacak şekilde genişletildi. Bu geliştirmelerin amacı, gecikmeli aydınlatmanın ileri gelen engellerinden biri olan saydamlık probleminin, gerçek zamanlı bir render motorunun sınırlarına bağlı kalarak saydamlık içeren sahnelerin görsel açıdan aslına uygunluğunu artırarak, kısmen üstesinden gelmekti. Işık kırılması ve saydam gölge destekleyen başka aydınlatma hatları ile beraber performans analizleri ve karşılaştırmaları yapıldı. Işık kırılması simule etmenin maliyeti ilave olarak 4 kaplama talimatı ve 13 aritmatik talimat olarak bulundu. Bu maliyet 60000 çokgen bulunan bir sahnede saniyedeki görüntü sa-

yısında %10 azalmaya sebep oldu. Test donanımında opak gölge yerine saydam gölge kullanmak saniyedeki görüntü sayısını 2 ışık içeren bir sahnede %13, 64 ışık içeren bir sahnede ise %34 azalttı. Bu çalışmada önerilen metod aynı etkileri destekleyen bir ileri aydınlatma motoruna göre, 8 ve 32 adet gölgeli ışık kullanıldığında sırasıyla en az %6 ve en çok %28 daha hızlı çalıştı.

Anahtar Kelimeler: Gecikmeli,Saydam,Gölge,Kırılma

To anyone who can enjoy a sunset in a game.

ACKNOWLEDGMENTS

I would like to thank ODTÜ-TSK MODSİMMER for providing me with a workplace during my studies and Simsoft Bilgisayar Teknolojileri Ltd. Şti. for providing me with additional test rigs. I would also like to thank my supervisor Prof. Dr. Veysi İşler for his continued support and guidance during the course of my education. Finally, I would like to extend my everlasting appreciation to my parents and my sister.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

G-buffer Geometry Buffer

GPU Graphics Processing Unit

2D Two Dimensional

3D Three Dimensional

RT Render Target

MRT Multiple Render Targets

HLSL High Level Shading Language

FPS Frames Per Second

RGB Red Green Blue

API Application Programming Interface

CHAPTER 1

INTRODUCTION

Whenever an interactive virtual environment is the subject in question, real-time feedback is always one of the most important restricting factors. Processing power intensive operations for sustaining an interactive virtual environment include and are not limited to physics simulations, artificial intelligence and rendering graphics. The general trade off in any interactive virtual environment is visual fidelity and realism versus performance. Video games, which commonly employ interactive virtual environments, try to provide the highest fidelity possible while still providing the player with real-time feedback.

As the demand for video games has been driving the speedy development of computer technologies, more and more algorithms have become available to be applied in real-time. However, while high visual fidelity techniques like ray tracing may be available real-time on experimental hardware setups, any technique to be used in games being developed for a larger target demographic, must target consumer level hardware.

One technique that became available on consumer level graphics processing units (GPU) is Deferred Shading. Deferred Shading, though not called by name, is first introduced as a concept along with a graphics processor design by Deering et al. in 1988 [2]. The idea is that every fragment is only shaded after going through depth testing successfully, which means that shading calculations are not done for fragments that are out of frame or are occluded by other objects. Disassociating shading operations from the geometry pass make using multiple lights much more efficient than using them in conventional forward shading, thereby overcoming one of the main limitations of forward shading which is limited number of lights.

The information needed for shading, such as surface normal vectors, are fed right through the pipeline proposed in [2] together with the rasterized fragments. However, in current incarnations of deferred shading the calculations required are done using multiple programmable shaders. Information such as fragment colors and normal vectors are passed to subsequent shaders, where lighting calculations are actually performed, through the use of a geometry buffer (G-buffer). G-buffer, introduced by Saito and Takahashi in 1990 [19], is a collection of images which contain the property of one surface per one pixel such as its normal, position, depth, color or reflectivity. Creating the G-buffer used to require multiple passes over the scene geometry since multiple images are

needed. This made deferred shading unusable in real-time applications until the introduction of multiple render targets (MRT) in GPUs. Using multiple render targets, multiple images can be output from a single programmable shader in a single geometry pass [14], which also means that the G-buffer can be created in a single geometry pass.

The widespread availability of MRT in GPUs made deferred shading a feasible rendering technique to be used in games. Notable game engines using deferred rendering techniques include Unity [23], CryENGINE 3 [13] and Unreal Engine 4 [24] while some games which employ deferred rendering techniques are Crysis 2 and Crysis 3 which use CryENGINE 3, Starcraft 2 [7], S.T.A.L.K.E.R [20] and Tabula Rasa [12].

Since the G-buffer can contain the property of only one surface per one pixel, handling transparency is problematic in deferred shading, as rendering transparency requires information from multiple surfaces per one pixel to blend properly. However, transparency is a part of the real world, such as in windows, glasses or bottles. Therefore, transparent surfaces must be available in a renderer aiming to simulate the real world. In many cases where deferred rendering techniques are used, such as in [7], [13], [12] and [20], transparent objects are handled in a separate forward renderer. However, this requires the need to maintain two renderers which can introduce lighting inconsistencies to the scene.

One way of handling transparent objects in deferred shading is using depth peeling introduced by Everitt et al. in 2001 [5]. In depth peeling, every layer of transparency with respect to the camera's view point is extracted to a separate image. For every layer to be extracted, a separate pass over scene geometry is required. Performing multiple passes over the scene geometry make this technique undesirable to be used in real time even with a forward renderer. To apply depth peeling to deferred shading, each layer must also have it's own G-buffer which would increase the already large memory imprint of deferred shading.

In 2009, Pangerl introduced a way to handle a single layer of transparency in deferred shading without any additional passes over scene geometry or additional rendering pipelines [15]. This is done by interlacing the transparent fragments closest to the viewer in the G-buffer. After shading calculations are done, the transparent lines are deinterlaced and are blended with the opaque lines. This technique both ensures light consistency in the scene and removes the need to maintain multiple rendering pipelines. This technique brings with it some disadvantages. Firstly, based on the interlacing method used, areas in the image containing transparent objects may suffer from lower resolution. Secondly, a single layer of transparency may look unnatural where multiple transparent objects partially occlude each other.

The aim of this study is to provide techniques to be used with deferred shading to increase the visual fidelity of scenes containing transparent surfaces while abiding with the constrains existing for a real-time renderer. This is done by incorporating refraction to a single layer of transparent surfaces and transparent shadows into the scene. No additional rendering pipelines are utilized and the techniques are implemented on the GPU using programmable shaders.

This study consists of four additional chapters. Firstly, Chapter 2 explains the illumination model used, then gives a more detailed explanation of deferred shading and its implementation. Finally, techniques related to transparency are discussed with previous works that incorporate transparency with deferred shading. Chapter 3 presents the details of the methods that are proposed for simulating refraction and creating transparent shadows. Chapter 4 first describes the implementations created for evaluation purposes. Later on Chapter 4 lists and discusses the results of the various evaluations performed, such as performance analysis of proposed techniques and implementation comparisons. Finally Chapter 5 summarizes the study and discuss possible future works.

CHAPTER 2

BACKGROUND AND RELATED WORKS

This chapter provides background information about shading and presents previous works that deal with both transparency and deferred shading. The first section explains the illumination method used as a basis in this study. In the the next section, an overview of deferred shading and methods that are used to implement it are given. The third section looks into transparency in computer graphics and recent works that integrate transparency techniques with deferred shading.

2.1 Illumination Model

To calculate the illumination along a surface, the later named Phong Reflection Model was introduced by Phong [16]. Phong states an illumination model developed to simulate real physical objects calculates illumination by using the characteristics of the light, the characteristics of the object and the position of the viewer. Therefore in this model a light has two components that contribute to the perceived intensity from a surface. The first is a diffuse component, which simulates the light that scatters in every direction while being reflected. This component constructs the main color perceived from the object. The second is a specular component, which simulates the light that is perfectly reflected, such as from a shiny metallic surface, in the direction of the viewer. This forms a highlight on the object that depends on the location of the viewer and the specular properties of the object. Equation 2.1 is the simplified form of the proposed illumination function used to calculate I_P, the perceived intensity for a single light at a point $\mathbf{P}$:

$$I_P = (k_d(\mathbf{N}.\mathbf{L}) + k_s(\mathbf{R}.\mathbf{V})^{n_s})I_L \tag{2.1}$$

In this equation, k_d and k_s are respectively the diffuse and the specular colors of the object and are dependent on the material of the object. $\mathbf{N}.\mathbf{L}$ is the dot product of $\mathbf{N}$, which is the normal vector at $\mathbf{P}$, and $\mathbf{L}$, which is the direction vector from $\mathbf{P}$ to the light source. Similarly, $\mathbf{R}.\mathbf{N}$ is the dot product of $\mathbf{R}$, which is the reflection vector of $\mathbf{L}$ with respect to $\mathbf{N}$, and $\mathbf{V}$, which is the viewing direction vector from $\mathbf{P}$ to the viewer . n_s is an exponent which models the specular characteristic of the material. Finally, I_L is the color and intensity of

the light. It should also be noted that an attenuation can also be applied to I_P to produce more realistic results. The attenuation for a point light, which is spherical and emits lights equally in every direction, can be calculated with a distance based attenuation coefficient. For a spot light, which emits light in a cone, the intensity can be decreased as the angle between $\mathbf{L}$ and spot light direction increases in addition to being affected by distance.

For multiple lights in the environment, I_P is calculated for each light and added together for the final light intensity at P. Since this reflection model is a local illumination model where only direct light response is calculated, the inter-surface diffuse reflections are not accounted for. To simulate this effect an ambient term can also be added. This can be modeled as $k_a I_a$ where k_a is the ambient property of the material based on its absorptive and reflective properties and I_a is the amount of total ambient light in the environment.

2.2 Shading

Shading, in the context of this study, is the collection of algorithms which are used to calculate the amount of light reflected by a surface from a light source, in the viewing direction, for example to a camera. In other words shading is the process which illuminates the objects in a virtual environment.

2.2.1 Rendering Pipeline

The implementations in this study are done using XNA Game Studio, which provides an easy to use content pipeline to manage models and textures while retaining access to the graphics pipeline [27] through the use of effects which can be customized with programmable shaders. XNA framework renders graphics using the Direct3D 9 graphics pipeline. The accessible portion of Direct 3D pipeline from XNA Game Studio is shown in Figure 2.1.

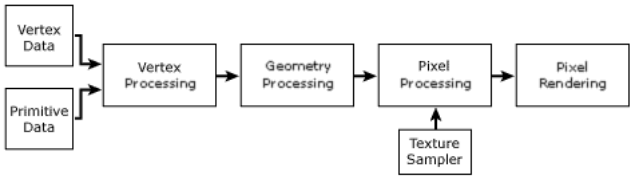

Figure 2.1: XNA Game Studio Rendering Pipeline (retrieved from [27])

To be drawn with this pipeline, anything that is to be drawn is defined in terms of primitives, such as points, lines or triangles. A primitive's vertices are set to the vertex buffer and these vertices' indices at the vertex buffer are set to the index buffer. A vertex in the vertex buffer can contain more information then just a vertex position. Depending on the vertex declaration, a vertex could contain other parameters such as texture coordinates or normal for that

vertex. Then, this data goes through vertex processing, geometry processing and fragment processing. These three processing blocks are made accessible to developers through HLSL which stands for High Level Shading Language for DirectX. HLSL is a C-like language that can be used to create programmable shaders, which are programs that will be used to render the primitives instead of the fixed function pipeline of Direct3D. Vertex shaders, geometry shaders and fragment shaders can be created for the above mentioned processing blocks using HLSL.

The vertex shader transforms each vertex from object space to screen space using the model, view and projection matrices which define object position and orientation and camera properties. Other operations can be performed in the vertex shader like performing cheaper but lower quality lighting based on vertex normal, or transforming the vertex normal to world space so lighting can be performed further down the pipeline. After this, the vertex shader sends the output to a geometry shader if it is defined. If a geometry shader is not defined, the output is sent to the rasterizer. These output vertices can also contain multiple parameters like texture coordinates or normals.

The geometry shader takes a primitive as input and can introduce new primitives or alter existing ones. While there are many other capabilities of the geometry shader, they are out of the context of this study and will not be discussed. The output from the geometry shader is sent to the rasterizer.

The rasterizer produces raster images for the primitives, which produces fragments for the primitives. For example, for a triangle primitive defined with three vertices, the rasterizer produces fragments on a 2D image that triangle would occupy. The produced fragments contain the same parameter types as the vertex output. The value of these parameters depend on the interpolation mode of the shader. If interpolation is disabled, the parameters of the first vertex of the primitive are used for all of the fragments created for the primitive. If interpolation is enabled, the values at the fragment are calculated with bilinear interpolation for triangle primitives and linear interpolation for lines, between the vertices of the primitive.

Each fragment produced by the rasterizer is processed by the fragment shader. The fragment shader decides the final color of a fragment before it is passed down further for depth testing. Whether this fragment ends up in the final output or not depends on this depth test. If a fragment passes depth testing, it is accepted and depending on the blending mode set, it will replace the existing fragment at the same screen space coordinates or be blended over it. Texture sampling, per-pixel lighting and many other things like filtering and post-processing can be performed in the fragment shader.

The output can either be sent to the frame buffer of the GPU to be displayed on the monitor or be written to a render target, which can be thought as a 2D image. Direct3D 9, however, also supports rendering to Multiple Render Targets (MRT) which means that fragment shaders can provide more than one output to these render targets with a singe pass over scene geometry.

2.2.2 Forward Shading

In conventional forward shading, lighting calculations for an object are performed directly in the geometry pass used to render that object. No immediate steps are taken once the objects are sent over to the GPU.

When Phong Reflection Model is applied to vertices in the vertex shader, flat shading occurs if interpolation is disabled. If interpolation is enabled the illumination effect observed is equivalent to Gouraud Shading introduced by Gouraud [8]. However if the normals can be interpolated from vertices as proposed by Phong in [16], than per-pixel lighting can be performed. While this process, now called Phong Shading, produces more accurate results when compared with Gouraud Shading, it is also more expensive. Phong Shading can be performed with HLSL when the normals for each vertex in the vertex shader are transformed to world space and included in the vertex shader output. If interpolation is enabled, the fragment shader input includes interpolated normals which can be used to perform per-pixel lighting. However, it should be noted that to calculate proper direction vectors, the world position of the fragment should also be available to the fragment shader. This can be done by sampling the screen coordinates and passing the depth from vertex shader (which would also be interpolated to the correct value) at that fragment and applying the reverse transformation matrix to transform screen space coordinates back into world space.

If multiple lights are desired with forward shading, either each of the lights' parameters must be passed to the GPU memory to be used by the shaders where the lights can be looped over to accumulate the intensities or multiple passes over the scene geometry is required where only the intensity of one light is calculated each pass and the intensities are accumulated using additive blending.

2.3 Deferred Shading

There are multiple drawbacks of using per-pixel lighting with forward shading. One is that lighting operations are performed for all fragments given to the fragment shader whether it would make it to the final output or not and this means there are redundant calculations which slow down the rendering process. Another drawback is, without using a technique like space partitioning, it is not possible to know which lights effect which objects and therefore even if the attenuation would cause the intensity to be imperceptible, the lighting operations are still performed, which again results in redundant lighting calculations.

Deferred shading is a technique where the lighting calculations for a pixel are separated from the geometry pass and therefore are only performed for pixels that are going to appear in the final image. Deferred shading is first introduced as a concept, though not called by name, in a graphics processor design by Deering et al. [2]. In [2], the authors propose a graphics processor where they handle rasterization differently with their own pipeline, mainly composing of a triangle processor followed by a normal vector shader. The triangle processors raster-

ize the transformed triangles, however instead of just preparing the pixels and calculating their depths, it also uses bilinear interpolation to calculate surface normals and viewing vectors at the pixels and propagates material information for the pixel. The triangle processor also performs depth testing while preparing pixels. When these processes are over, the data is propagated to normal vector shaders, which perform per-pixel Phong Shading using the interpolated data.

Today, however, deferred shading does not require a dedicated hardware to be implemented. A clear implementation was showcased by Hargreaves and Harris in 2004 [9] and this is explained below.

First a geometry pass is performed on the whole scene. The geometry pass is used to create multiple images, all of which contain information needed for the lighting calculations such as colors, normals and positions, and this collection of images is generally called a **geometry buffer** or G-buffer. The G-buffer was introduced by Saito and Takahashi in 1990 [19]. In [19], the authors treat G-buffer as an intermediate step which later can be used for numerous image based post-processes, including but not limited to shading.

After the G-buffer is created, each light is rendered as a 3D model into another image with additive blending enabled. For example a sphere and a cone would be used for point and spot lights respectively. Drawing a light as a 3D model with the same projection and camera setup produces pixels in the screen coordinates where the volume of that light would appear. Since any object in that volume would also generate pixels at the same screen coordinates, in conjunction with a depth-test, it can assumed the pixels for that object are inside the light volumes accumulated to that pixel. Therefore the values at the G-buffer at the same screen coordinates can be used to calculate illumination at a pixel. However, while the illumination is being calculated the object colors k_d and k_s are omitted, which are already in the G-buffer. This image is called **light accumulation buffer**.

Finally a full screen post-processing pass is done using the G-buffer and the light accumulation buffer as input textures. These textures are overlapped and combined to complete the illumination calculation and the final image is created. A step by step tutorial for implementing Deferred Shading with XNA Game Studio can be found in [3].

2.4 Transparency

When looked through transparent objects, these objects provide a mix of their own color and the color of whatever objects are behind. An image compositing technique is required to simulate this effect in a rendering pipeline, where the drawn transparent triangles can mix their own colors with whatever was behind. One such technique is described by Porter and Duff in 1984 [17]. In their study, Porter and Duff use a fourth channel, called the alpha channel, to be included with an image for every pixel in addition to the traditional channels of red, green and blue (RGB). This value at the alpha channel would then represent

the permeability of the pixel. Higher alpha values at a pixel mean, that pixel is more opaque and it contributes more to the final image while being mixed with pixels behind it. The proposed compositing function shown in Equation 2.2.

$$C = C_A * F_A + C_B * F_B \tag{2.2}$$

In Equation 2.2, C_A and C_B are respectively the colors of the pixels being composited A and B in represented with RGB channels. F_A and F_B are fractions based on the compositing technique being used, utilizing the alpha values of A and B, α_A and α_B respectively. To simulate the effect of A being over B with respect to the viewer, F_A is defined as α_A and F_B is defined as $(1 - \alpha_A)$.

This method of compositing, commonly known as alpha blending, is very widespread and readily available on GPUs through the use of a graphics application programming interface (API) such as OpenGL or Direct3D. When alpha blending is enabled, a pixel produced that passes depth testing (or is otherwise accepted) enters the alpha blending function as the source value, while the already existing value at the frame buffer or render target is the destination value.

Assuming the source pixel is in front of the destination pixel with respect to the viewer and using the F_A and F_B fractions defined above for overlapping surfaces, the alpha blending function to simulate a transparent pixel in front of an already existing pixel becomes:

$$C_d{}' = C_s * \alpha_s + C_d * (1 - \alpha_s) \tag{2.3}$$

In Equation 2.3, C_s and C_d represent the source and destination pixels' colors respectively while α_s is the alpha value, or the level of opacity, of the source pixel. $C_d{}'$ is the composited color to be overwritten to the destination pixel.

Most rendering pipelines use procedures similar to the ones shown in [22] to simulate transparency for traditional transparent surfaces such as glass. As stated in [22], since a transparent object's back facing polygons may also be visible through it's front facing polygons, they must also be drawn without being culled. Another issue mentioned is that the most visually accurate results are reached when objects being drawn are done so using a back-to-front order with respect to the viewer, so that any new transparent layer only blends on top of everything that has already blended properly. However, just depth sorting objects may not be enough since the primitives defined for the objects are in many cases not sorted with respect to the viewpoint. Even sorting individual polygons may not be enough since a correct ordering may not be found for overlapping and intersecting polygons such as shown in Figure 2.2.

In any case, the transparency techniques described in [22] cannot be used directly with a deferred shader, since before being blended any transparent pixel must be shaded with its own parameters such as normals while a deferred shader does not do this during the geometry pass where alpha blending is done. Alpha blending also can not be performed later on in deferred shading because the created G-buffer only contains information of only one surface per one pixel.

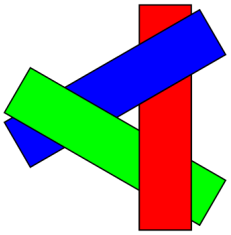

Figure 2.2: Overlapping Polygons

2.4.1 Depth-Peeling

Depth-peeling is a method described by Everitt in 2001 [5] where multiple layers of objects with respect to the viewpoint is peeled into separate textures, which can later be blended together to produce pixel-ordered accurate transparency results. To achieve this, first a pass over the scene geometry is performed with depth testing and writing enabled and set to accept the fragments closest to the camera. The depths for the accepted fragments are written to a depth map, which is implemented through a hardware shadow map. For each of the subsequent passes over the scene geometry, the depth buffer is cleared and in addition to conventional depth test accepting nearest fragments, the fragments are also checked to be farther away from the previously prepared depth map. Therefore any fragment that has been rendered previously is peeled away and rejected with the test against the depth map, allowing a second nearest layer of fragments to be accepted. The depths from these subsequent passes are also written to a depth map, which will be used to peel away an additional layer in the next pass.

With this technique each pass over the scene geometry generates a layer image farther away from the viewpoint in order, which can be blended together by rendering each layer image as a viewport-sized textured quad (a full-screen quad) and enabling alpha blending.

Depth peeling is compatible with deferred shading, however instead of generating a colored image for each layer, a complete G-buffer must be generated for each layer and the final composition phase of deferred shading for each layer must be performed by rendering a viewport-sized textured. While additional geometry passes over the scene geometry required for depth peeling are already costly, the additional memory imprint brought on by maintaining all the G-buffers makes this technique impractical and too expensive for the currently widespread hardware.

2.4.2 Transparency and Deferred Rendering

Many previous works, including commercial games like Starcraft 2 [7], S.T.A.L.K.E.R [20] and Tabula Rasa [12] or game engines like Unity [23] and

CryENGINE 3 [13], have always used two separate rendering pipelines to support transparent objects. One renderer where opaque objects are rendered using deferred techniques, followed by a forward renderer which draws transparent objects over the scene already illuminated by deferred rendering.

In 2009 a technique is proposed by Pangerl [15] which basically compacts a layer of opacity and a layer of transparency together into a single G-buffer, so that the data extracted from this buffer can be used in the final composition stage of deferred shading. During the G-buffer generation phase required for deferred shading, any calculation relevant to transparent surfaces is only done on the odd lines of the images. This is achieved by using a programmable fragment shader to interlace transparent and opaque pixels together by rejecting pixels when the fragment being processed is transparent and its vertical screen coordinate corresponds to an even line.

After the G-buffer is created with interlaced transparent lines, the lights are accumulated in the light buffer normally. In the final composition phase of the deferred renderer, each fragment processing a sample from the G-buffer also samples an additional coordinate from one line higher. If the fragment would originally belong to a transparent object on the final image, one of these samples belongs to that transparent object and the other sample belongs to the opaque object directly behind it. The illumination values of both samples are applied from their coordinates at the light accumulation buffer and and the illuminated samples are blended together using the alpha value of the transparent sample. This effectively creates a single layer of transparency blended over the opaque object. One drawback of the algorithm is the vertical resolution is halved in the image at places where transparent objects exist.

2.4.3 Refraction

While refraction can be handled inherently in a ray tracer by generating refracted rays using Fresnel equations and tracing them subsequently, in forward shading these refraction needs to be simulated.

A technique proposed by Sousa in 2005 [21] applies a perturbation to the coordinates of the fragments used in blending the transparent surfaces with the background. This 2D image based perturbation is calculated from the normal of the transparent fragment being blended and a small coefficient. This technique requires the opaque objects in the scene to be rendered to a texture, which denoted as S, in order to sample this texture based on coordinates which can be perturbed and sampled while the transparent objects are being rendered.

Since every opaque object in the scene is already rendered, the sample from the perturbed coordinate may come from an object in front of the transparent surface. This creates a leakage where artifacts from objects closer to the viewpoint than the transparent surface appears on the transparent surface. This is handled by utilizing the alpha channel of S by putting 1 where opaque objects exist and 0 where transparent objects exist, which creates a kind of an alpha-mask. Later while the refraction is being applied, the perturbation is reverted to the original

coordinate if the value of S at the perturbed coordinate is 1. This way perturbed coordinates are only applied if the sample is behind a refractive object.

2.5 Shadow Mapping

The basic shading process used in forward and deferred rendering do not take into account whether the fragment being shaded has direct exposure from a light or is occluded by some other object. Since no occlusion from lights is taken into account, every object is lit as if it has direct exposure from every light on its surfaces and as such no shadows appear on the scene.

Shadow Mapping introduced by William [26] is a technique to include shadows in scenes. In the technique proposed in [26] the occlusion from a light for a fragment being lit is calculated from a texture which contains the depth values of objects closest to that light. This texture, called the shadow map, is generated by rendering the scene geometry from the light's point of view with depth testing enabled only outputting depth values. When a fragment is being shaded by a light, the shadow map generated for that light is sampled at the coordinate where that fragment would appear from the light's point of view. This is done by applying a reverse transformation matrix to the fragment, which converts the fragment's coordinates back to world-space, and applying the transformation matrix used for generating the shadow map for that light to the world coordinates of the fragment. If the depth sampled is less than the fragment's depth with respect to the light, it means that the fragment is occluded by another object and as such is not lit. Otherwise the fragment is from the object closest to the light and is lit normally. It should be noted that the quality of the shadows created highly depend on the resolution of the shadow maps and the sampling method used. Lower resolution shadow maps cause aliased shadows and the default sampling produces hard shadows. A technique where multiple points are sampled and averaged in the shadow map can be used to anti-alias the shadows and create softer shadows [18], [6].

To generate shadow maps different kind of projections are used for different light types. For a directional light, an orthographic projection is used which causes all the shadows to appear in the same direction. For spot lights, a perspective projection where the viewing angle is equal to the spot angle is used. For point lights however, no single traditional projection method can cover all of the directions around the point light. Multiple methods for handling point light shadows are discussed in [1]. While a spherical projection can be used for point lights, this distorts the shadow map too much and lowers the resolution. A cube map can be produced from six shadow maps which means the scene is rendered six times with perspective projection for point lights, however this is too inefficient for real-time applications. The method proposed in [1] uses two paraboloid shadow maps for each half of the sphere of a point light, which can be generated in two passes over scene geometry per point light.

2.5.1 Transparent Shadows

Default implementation of shadow maps do not consider transparent objects in the scene and any shadow created is colorless (black if there are no other lights in the scene). However accounting for all of the transparent objects in the scene would be inefficient as this would require several shadow maps which contained both depth and color values for each layer of transparency from the light's point of view.

In [7], only a single layer of transparent shadows are handled. In the method proposed in [7], each light has one depth map for opaque objects, one depth map for the transparent objects closest to the light and one color map where the colors of the transparent objects are accumulated as light filters. The color map is created by rendering the transparent objects sorted front-to-back where each transparent object rendered filters some of the light based on its color. Any depth farther then the opaque depth map has complete shadows. Any depth less than the opaque depth map and greater than the transparent depth map is lit by the light filtered by all transparent objects in between. Depths in front of both depth maps are lit normally. In this approach transparent objects can only be shadowed by opaque objects while they are not affected by other transparent objects' shadows.

CHAPTER 3

PROPOSED METHOD

In this chapter the proposed method is explained for increasing the visual fidelity of scenes rendered with deferred shading that include a single layer of transparency, by adding shadows of transparent objects and simulating refraction of these transparent objects.

3.1 G-buffer

The setup for a deferred shader with transparency starts by choosing what to include in the G-buffer and selecting storage formats. Since transparency is not supported in default deferred renderers, alpha values defined in the textures of materials used by the objects are not included in the G-buffer, however to support a single layer of transparency, the alpha value is also included in the G-buffer. The complete G-buffer setup and the render target formats used for the four render targets are shown in Table 3.1.

Table3.1: G-Buffer Render Targets

RenderTarget Formats	**Stored Values**			
RT 0: R8G8B8A8	Diff. Red(8)	Diff. Green(8)	Diff. Blue(8)	Trans. Alpha(8)
RT 1: R10G10B10A2	Normal X(10)	Normal Y(10)	Normal Z(10)	unused(2)
RT 2: R8G8B8A8	Spec. Intensity(8)	Spec. Power(8)	unused(8)	unused(8)
RT 3: R32	Depth(32)			

To maintain support in a wider arrangement of graphic cards, all render targets use the same bit depth which is 32-bits. 64-bits render targets can also be used, for example to increase color depth and including additional information for other effects, but in this implementation they are avoided to minimize G-buffer size and reduce the memory footprint.

In Table 3.1, RT 0 is the diffuse component of the G-buffer which stores the diffuse response of the fragment with its transparency value. RT 1 stores the normals of the fragment, using 10-bits for each of the RGB channels allows for higher precision normals to be used during specular component resolution in the lighting phase [9]. RT 2 stores the specular properties of the fragment, which are sampled from a specular map supplied by the model artist. Any specular high-

lights are considered white. This makes it possible to stored specular intensity response of a fragment in a single 8-bit channel as it is shown in [9]. The specular map can also contain specular power for the fragment and it will be stored in a secondary 8-bit channel in RT 2 [9]. Lastly, RT 3 stores the screen space depth of the fragment. To possess sufficient precision, a single 32-bit channel render target is used.

3.1.1 G-buffer Creation

A vertex shader and a fragment shader are used to produce the G-buffer. The render target of the graphics API is set to multiple render targets, to the ones shown in Table 3.1. Depth tests and writes are enabled to accept fragments closer to the camera while blending is disabled. The whole scene geometry is rendered using this effect to create the G-buffer.

The vertex shader accepts vertex positions, normals, tangents, binormals and texture coordinates. The vertex normals, tangents and binormals are transformed to world space and are combined in a matrix which can be used later in the fragment shader to transform the supplied normals in the normal map of the object to world space. The vertex positions are transformed into screen space. The vertex shader outputs the transformed vertex position, the depth at this position, texture coordinates and the calculated normal matrix.

After rasterization with bilinear interpolation enabled, the fragment shader accepts the output format of the vertex shader in addition to the fragment position in screen coordinates (which can be supplied to the shader by the graphics API). The fragment shader samples the diffuse color texture at the given texture coordinates. If the sampled color's alpha value is less than 1, it means the fragment belongs to a transparent object. Transparent fragments are rejected if they are on an even line on the G-buffer. For other fragments, their normal and specular maps are sampled. The normals are transformed to world space using the normal matrix calculated by the vertex shader. Fragment color, transparency alpha value, depth, transformed normals and finally specular intensity and power are output to the G-buffer in the format shown in Table 3.1.

Once the G-buffer is completed, it contains any information needed for lighting and blending a single layer of transparent objects on the odd lines where transparent objects exist. The information for lighting opaque objects are everywhere else on the G-buffer.

3.2 Light Accumulation Buffer

Another pair of shaders are used to accumulate the lighting information for multiple lights in a light accumulation buffer. This buffer outputs to a single render target shown in Table 3.2. Three kinds of lights are supported; spot, point and directional while only spot lights can cast shadows to simplify the implementation. Spot and point lights are modeled by cones and spheres, respectively,

which are scaled, positioned and rotated depending on light parameters and orientations. Since directional lights, such as sun or moon, traditionally effect everything in the scene, a full screen quad polygon is drawn for directional lights so that every fragment on the scene goes through directional light calculation as it is done in [11].

Table3.2: Light Accumulation Buffer

RenderTarget Formats	**Stored Values**			
RT 0: R8G8B8A8	Red Response	Green Response	Blue Response	Specular Response

The render target for the graphic API is set to the one shown in Table 3.2. Depth writes and tests are disabled. Blending is set to additive blending with both source and destination blending fractions set to 1. This allows multiple lights to be blended by accumulating their effects on to the scene. It should be noted that for point and spot lights, cull settings are set to cull front faces for light volumes encompassing the camera position while they are set to cull back faces otherwise as shown in [9].

3.2.1 Transparent Shadows in Deferred Shading

As with conventional shadow mapping, for every shadow casting spot light, the scene is rendered once from that light's point of view. However, while creating the shadow map, instead of using a single depth map texture as an output, two textures are used. This is again achieved by using MRT, with the render targets chosen shown in Table 3.3.

Table3.3: Render Targets of Shadow Maps with Transparency

RenderTarget Formats	**Stored Values**			
RT 0: R32	Depth(32)			
RT 1: R8B8G8A8	Diff. Red(8)	Diff. Green(8)	Diff. Blue(8)	Trans. Alpha(8)

In addition to the depth map (RT 0), a color texture (RT 1) is also filled to include the color of the transparent object that the light will pass through. However to retain the complete light occluding effect of an opaque object while performing only a single pass over scene geometry per light, the depth where the light would be totally occluded and the depth where the light would pass through a transparent objects before illuminating the one behind it are compacted into a single depth map. This is achieved with the interlacing concept shown in [15].

To render the shadow maps another pair of vertex and fragment shaders are used. The render target of the graphics API is set to multiple render targets, to the ones shown in Table 3.3. Depth tests and writes are enabled to accept fragments closer to the camera while blending is disabled.

While rendering the scene geometry from the light's point of view, in the fragment shader, if the fragment being processed is transparent and its the screen

coordinates correspond to an even line, the fragment is rejected. Otherwise the fragment is accepted depending on the depth test and the transparent fragment's color is written to the color part of the shadow map while its depth is written to the depth part. Therefore, wherever in the shadow map there is a transparent object, depth and color information belonging to it are written in the odd lines of the shadow map while the lines directly below would contain the depth of the first opaque object behind it.

With this method the shadow map for a spot light, containing depth and color, can be produced with a single pass over the whole scene geometry.

During the light accumulation phase for a light, the fragment being processed is transformed back into the world coordinates and then transformed into the light's screen space coordinates where it can be compared against the shadow map of that light. Let the depth of a fragment P, with respect to the light illuminating it be denoted by d_P. For every fragment in the fragment shader, the light's shadow map is sampled at two coordinates to determine the fragment's occlusion from the light; once at the fragment's position in the light's screen space coordinates and once at the coordinates directly one line below.

The sample with the minimal alpha value is chosen as the transparent fragment potentially occluding P while the sample with the maximum alpha value is chosen as the opaque fragment potentially occluding P.

$$I_{L_{shadowed}} = \begin{cases} 0, & \text{if } d_{maxA} < d_P \\ min(I_L, C_{minA}) * (1 - \alpha_{minA}), & \text{if } d_{minA} < d_P \leq d_{maxA} \\ I_L, & \text{otherwise} \end{cases} \tag{3.1}$$

In equation 3.1, $I_{L_{shadowed}}$ is the color and intensity which will be used to calculate the illumination while I_L is the original color of the light. d_{maxA} and d_{minA} are the depths of the shadow map samples with the maximum and the minimum alpha values respectively. $min(I_L, C_{minA})$ denotes the light filtering function of the transparent surface where the minimum value is taken for each of red, blue and green components of the light and the color of the sample with minimal alpha which would belong to a transparent object, as shown in Equation 3.2. Finally α_{minA} is the alpha value of the sample with the minimum alpha value, which would belong to a transparent object and denote its transparency.

$$min(I_L, C_{minA}) = (min(R_{I_L}, R_{C_{minA}}), min(G_{I_L}, G_{C_{minA}}), min(B_{I_L}, B_{C_{minA}}) \tag{3.2}$$

While the light accumulation buffer's RGB components output light colors, it's alpha channel contains the specular intensity calculated using Phong Reflection Model shown in Section 2.1.

For the shadow maps of all lights in the scene, only a single collection of two render targets is sufficient, since any light uses only its own shadow map. There-

fore, the shadow mapping preparation for a light is performed right before the accumulation rendering for that light. The shadow map is cleared before another light's shadow maps are being calculated.

3.3 Refraction in Deferred Shading

To simulate refraction in deferred shading, a method similar to the one proposed in [21] is used where the transparent objects sample opaque fragments behind them with a perturbation on sample coordinates based on the transparent object's normal vectors. The separate passes over refractive meshes and opaque meshes are not performed in this method, it should also be mentioned that there is no need for the alpha stencil which would be used to prevent leakage from opaque objects in front of refractive surfaces.

The G-buffer generation phase does not need to be changed to introduce refraction into the deferred rendering pipeline with a single layer of transparency. Any calculation required is done in the final composition phase with the fragment shader. This is done with another full-screen quad drawn with a pair of vertex and fragment shaders to process every fragment in the scene, so they can be blended and lit properly.

In the fragment shader, the G-buffer is sampled at both the coordinate of the fragment being processed and at the coordinate one line above it. Let the sample with minimal alpha value (which belongs to a transparent object) be denoted by S_t and the sample with maximum alpha value (which belongs to the background S_b). The normal of S_t is sampled from the G-buffer and transformed to screen space. The x and y coordinates of the screen-space normal is multiplied by a coefficient and added to the original fragment coordinate as it is done in [21] to get a 2D perturbation. The resulting coordinates are the perturbed coordinates, denoted by P, to sample for an opaque object which can be used to simulate refraction. However the sample at P can also be transparent since G-buffer contains transparent objects on odd lines. If this is the case, G-buffer is sampled once more at one line above P, which is guaranteed to be an opaque sample. Let the final perturbed opaque sample be denoted by P_o. At this stage, to prevent the leakage effect mentioned in [21] which occurs when the perturbed coordinates are on an opaque object in front of the refractive surface, the depths are sampled at S_t and P_o. If S_t is behind P_o, P_o is discarded since the perturbed coordinate is in front of the refractive surface and S_t is blended with the original background sample S_b. Otherwise P_o is an appropriate perturbed coordinate for refraction and S_t is blended with P_o.

After the final opaque coordinate to be blended is decided, the light accumulation buffer is sampled at this coordinate in addition to being sampled at the transparent coordinate. Where C_o and I_o are the diffuse color and light accumulated at the opaque coordinate and C_t and I_t are the diffuse color and light accumulated at the transparent coordinate, the final viewed illumination, I_F, can be calculated with Equation 3.3 where α_t is the alpha value which denotes the transparency at the transparent fragment at the G-buffer.

$$I_F = (C_o * I_o) * (1 - \alpha_t) + (C_t * I_t) * \alpha_t \tag{3.3}$$

This method introduces refraction into a deferred renderer with a single layer of transparency without interfering with the G-buffer generation and making use of already existing information.

CHAPTER 4

RESULTS AND DISCUSSIONS

In this chapter, first, the implementation of the proposed method is discussed. Later a summary to the implementations of a forward renderer and a deferred renderer with depth peeling is given. This is followed by the analysis of the proposed method's performance impact. After that a comparison of the proposed method with the alternate implementations is presented. Finally, the performance of the proposed method in scenes with different levels of transparency and different GPUs are analyzed.

4.1 Implementations

This section goes over different implementations of the effects proposed in Chapter 3. In addition to the implementation of the proposed method, a deferred renderer with depth peeling and a forward renderer with multi-pass lighting are also implemented for evaluation purposes. Both of these implementations support a single layer of transparency and refraction.

The Table 4.1 shows the render target usage of each of the implemented renderers, which directly affects the memory imprint of the algorithm. It should be noted that the extra render target required for simple shadows or the two extra render targets required for transparent shadows are not included in this table.

Table4.1: Number of separate render targets

Rendering Method	**Number of Render Targets**
Proposed Method	5
Deferred Peeling	10
Forward Rendering (w depth)	3
Forward Rendering (w/o depth)	2

4.1.1 Implementation of Proposed Method

As stated in subsection 2.2.1, the implementations of the proposed methods in this study are done using XNA Game Studio for maintaining assets like models

and communication with the graphics API, which in turn uses Direct3D 9 and HLSL.

The proposed method in Chapter 3 uses three different kinds of render target formats, which are R8G8B8A8, R10G10B10A2 and R32. The XNA Game Studio equivalent for these render target formats are SurfaceFormat.Color, SurfaceFormat.Rgba1010102 and SurfaceFormat.Single, respectively.

To import the models into the renderer, the custom content pipeline described in [3] is used. To create the G-buffer described in Section 3.1, an HLSL effect is created which implements the vertex shader and the fragment shader. The G-buffer is set as multiple render targets with *SetRenderTargets* method of XNA. The screen space coordinates required by the fragment shader are supplied by HLSL with the VPOS semantic. To reject fragments in the fragment shader while interlacing, the *clip* function of HLSL is used.

The diffuse and normal components of the G-buffer is shown in Figures 4.1, 4.2, respectively. The interlaced transparent objects can be viewed in both of these figures. A clear depth map is white in this G-buffer setup. Since SurfaceFormat.Single provides 32-bits for a single red channel, only the red value changes in the depth map. Closer fragments have lower red values, therefore closer objects appear in shades of cyan in the depth map. Example depth component of G-buffer is shown in Figure 4.3. (screenshot is from a different perspective than Figures 4.1 and 4.2)

After the G-buffer is created. The lighting phase begins, which is performed in a separate HLSL effect. For each spot light, first its shadow map is created as described in Subsection 3.2.1, followed by drawing a cone model to the scene transformed according to the spot light's properties using the methods described in Section 3.2. The shadow map created is sampled and tested against in the fragment shader of the effect which was used to draw the cone. The point lights are drawn as spheres, however their shadows are not implemented for the sake of simplicity. Point lights use a separate fragment shader since the attenuation formula is different than a spot light's and no shadow maps are sampled. The light accumulation buffer acquired after the lighting phase ends can be seen in Figure 4.5. The specular component is not visible in this image, since it is in the alpha channel and will be added on later in the combination phase.

It should be noted that while sampling a render target produced by XNA Game Studio which uses Direct3D 9, such as the shadow maps or the G-buffer, the screen coordinates in the fragment shader can not be used directly in the fragment shader. A step must be taken to correctly map texels to pixels which is described in [4]. This is especially important in this implementation since the deinterlacing requires precise mapping to work correctly.

After the lighting phase ends, a full-screen quad final combination pass is performed with another HLSL shader which implements the refraction and deinterlacing described in Section 3.3. The final image produced can be seen in Figure 4.6.

Figure 4.1: Example Diffuse Component of G-buffer

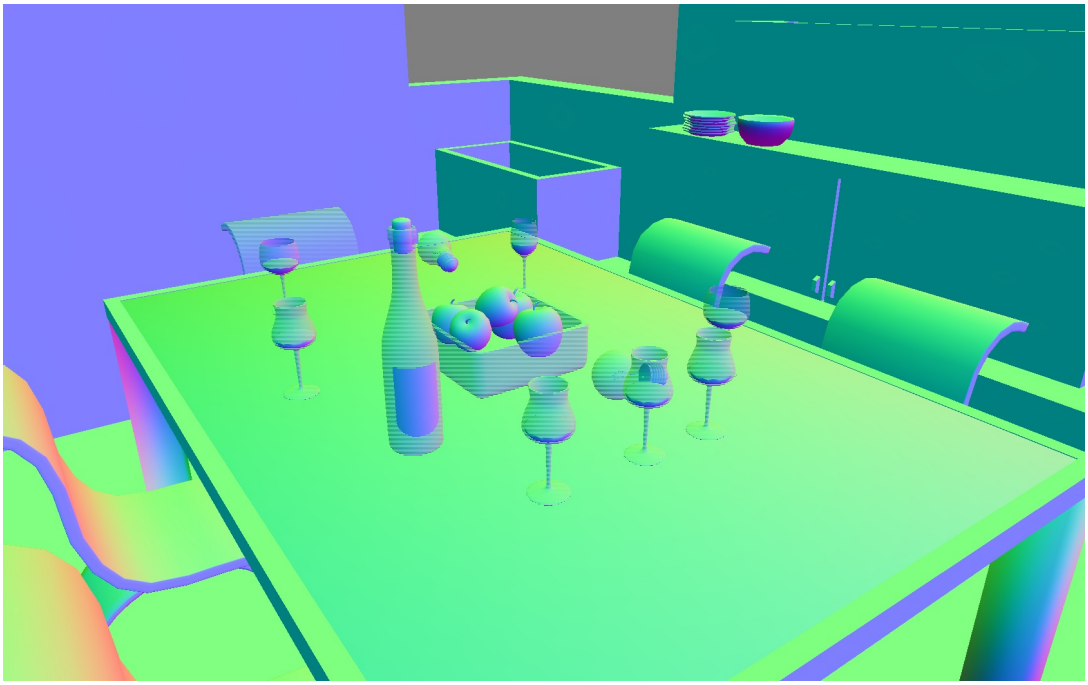

Figure 4.2: Example Normal Component of G-buffer

Figure 4.3: Example Depth Component of G-buffer

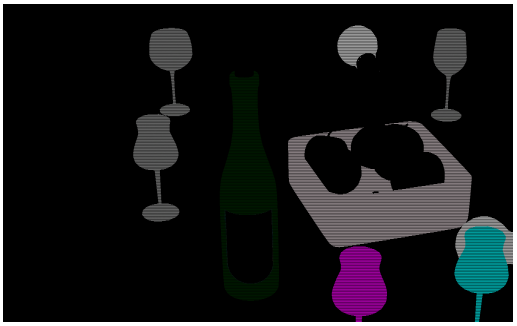

Figure 4.4: Example Color Component of the Transparent Shadow Map

4.1.2 Implementation of Deferred Depth Peeling

In order to generate visually equivalent results, only a single extra layer is peeled and is later used to combine a properly blended image. To illuminate both layers separately, this implementation uses two of every render target shown in Figure 3.1, corresponding to a separate G-Buffer for each layer.

The G-buffer for the first layer (closest to the viewer) is created similarly as explained in Section 3.1. The only difference is that no interlacing method is used, which means that this G-buffer will contain transparent objects without any loss in quality where ever they exist.

The depth component in the first G-buffer is used to peel off the first layer of objects in the scene. This is done by sampling the depth component of the first G-buffer while rendering the second G-buffer. With depth testing and writing enabled, this generates the G-buffer for a second layer of objects which were occluded by the first layer. Any transparency in the second G-buffer is rejected in the second G-buffer, which ensures only a single layer of transparency will be processed, to produce the same result with the main method proposed in Chapter 3.

During the light accumulation phase, separate light accumulation buffers are accumulated for each G-buffer. After rendering the shadow map for a light with the methods discussed in Subsection 3.2.1, this map is sampled while rendering the light for both light accumulation buffers. After the light accumulation phase is finished, the light accumulation buffers are applied to the diffuse components of the corresponding G-buffer for both layers. This generates two properly lit layers of objects, where one contains transparent objects and the other contains objects occluded by the first layer.

Finally to simulate refraction, both lit layers are passed to a full-screen quad drawn with an HLSL effect. This effect uses a similar perturbation method to the one discussed in Section 3.3. The correct perturbation is decided by sampling depths from each of the G-buffers and consequently blended, which results with the final image.

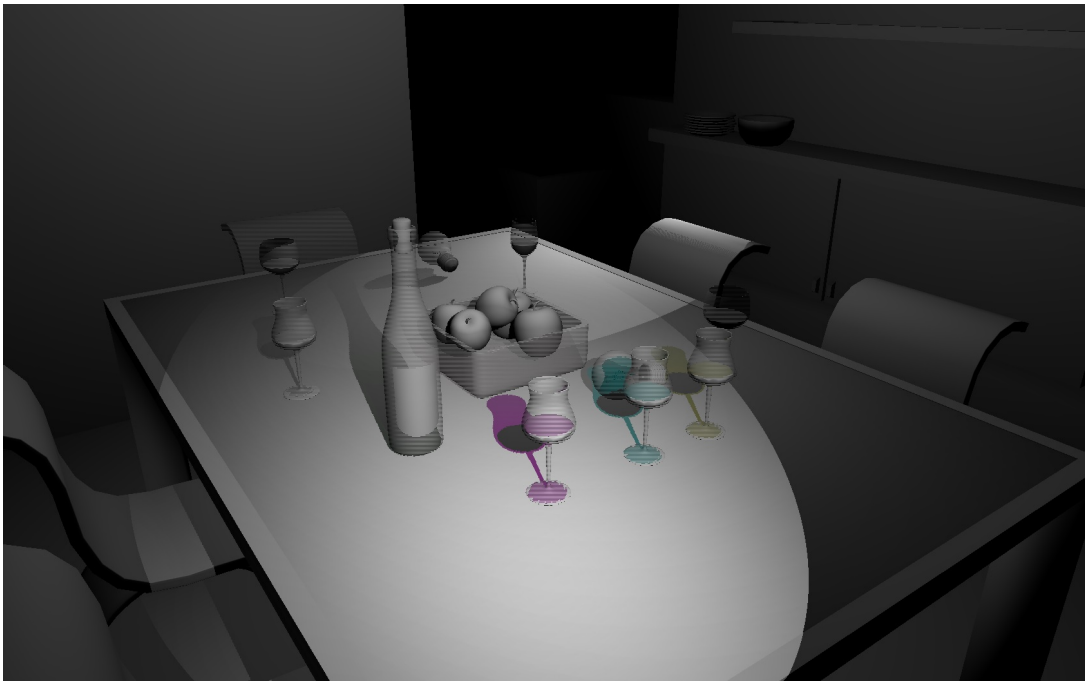

Figure 4.5: Example Light Accumulation Buffer

Figure 4.6: Example Final Composed Image

4.1.3 Implementation of Forward Rendering

The implementation of a forward renderer with multi-pass lighting supporting transparency comes with a dilemma. The problem is that accurate transparency blending requires the opaque objects in the scene to be completely rendered before any transparent objects are rendered. This presents with two options; either the shadow maps for each light have to be saved after being used to illuminate the opaque objects, so that they can also be used later while illuminating the transparent objects, or they have to be regenerated for each light while illuminating transparent objects. While the first option would increase the memory imprint significantly while using multiple lights, the second option effectively doubles the geometry passes performed for generating shadow maps.

Instead of using either of the previous options, the implementation here again makes use of the interlacing technique explained in Chapter 3. For each lighting pass, opaque objects and transparent objects are rendered simultaneously without discrimination, while rejecting the fragments belonging to transparent objects if they appear on the even lines of the image. This enables the use of only a single render target for all of the lights without regenerating any of them; however, it also brings the necessity for the fragment normals to be included in a separate render target, which can be acquired in the ambient light pass, which can be used to generate perturbed sampling coordinates while a full-screen quad is drawn to simulate refraction and perform blending in a blending pass.

To utilize the correct perturbation while simulating refraction, both the alpha-masking technique proposed in [21] and the depth comparison method discussed in Section 3.3 are implemented for evaluation purposes.

In the first implementation, in contrast to the method proposed in [21], no separate alpha-mask is generated beforehand with an extra geometry pass over refractive geometry and instead the already existing alpha information, acquired while deinterlacing the illuminated image, is used to determine if the perturbed coordinate is accurate.

The second implementation requires the fragment depths to be generated in the ambient pass and later included in the blending pass which can be used to determine if the perturbed coordinates are accurate.

With both of these implementations, the deinterlacing technique is similar to the one discussed in Section 3.3.

4.1.4 Refraction Error Reduction

While the forward renderer without depth sampling is expected to be faster than the forward renderer with depth sampling, two kinds of errors are visible in the images produced when a depth map is not used. Both errors can be seen in Figure 4.7.

In the technique explained in Subsection 2.4.3, perturbed coordinates are only

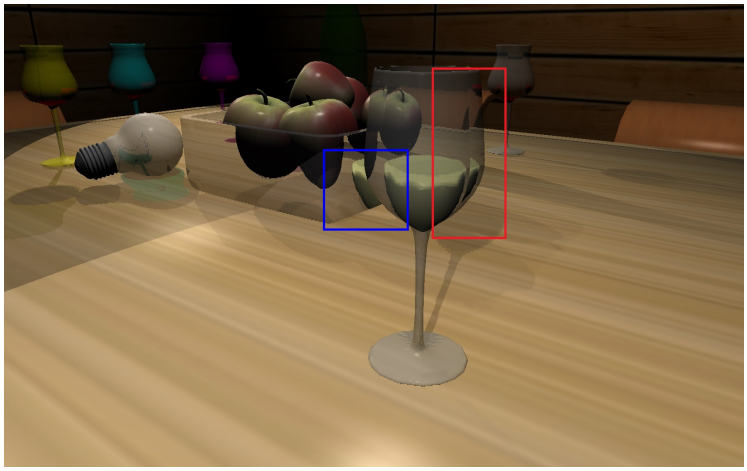

Figure 4.7: Composed Image With Refraction Errors

Figure 4.8: Correctly Composed Image

accepted if they appear in the area of any refractive surface and always rejected otherwise. The error outlined in blue happens when the opaque fragment sampled at the perturbed coordinate is accepted because it appears in the area of a refractive surface while it actually is in front of the refractive fragment and should have been rejected. The error in red happens when the opaque fragment sampled at the perturbed coordinate does not appear in the area of a refractive surface while it is actually behind the refractive fragment and could have been accepted.

Using a depth map in the second forward renderer eliminated both of these errors and produced the correctly composed image shown in Figure 4.8.

4.2 Performance Analysis of Effects and Comparison of Techniques

All of the implementations are tested in a single scene containing approximately 60000 polygons. The testing machine ran on an Intel i7 4770K processor without overclocking, containing 8 GB of RAM and a Geforce GTX 760 with 2 GB of RAM as its graphics card. All of the tests were performed using 1920 by 1080 resolution.

Table4.2: Average FPS during test course with/without refraction

# of Lights	No Effects	Refraction
0	564 FPS	499 FPS
2	490 FPS	440 FPS
4	438 FPS	397 FPS
8	364 FPS	338 FPS
16	331 FPS	306 FPS
32	283 FPS	265 FPS
64	148 FPS	142 FPS

Table4.3: Average FPS during test course with with opaque shadows or transparent shadows

# of Lights	Opaque	Transparent
2	394 FPS	343 FPS
4	311 FPS	254 FPS
8	226 FPS	173 FPS
16	163 FPS	114 FPS
32	108 FPS	71 FPS
64	53 FPS	35 FPS

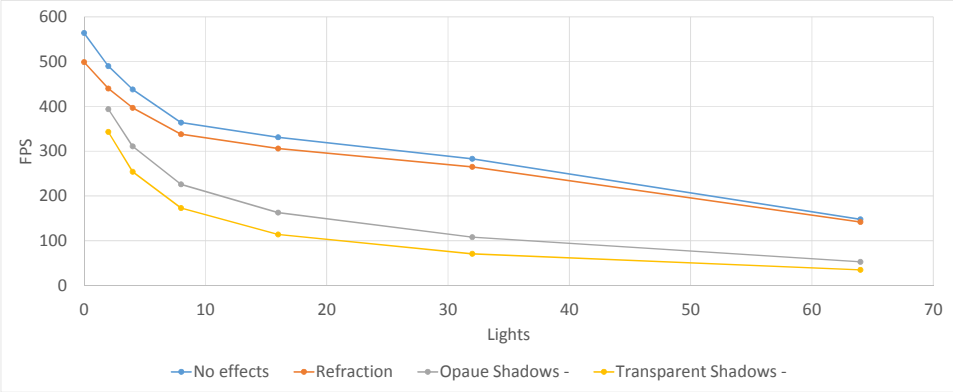

Figure 4.9: Chart of Table 4.2 and Table 4.3

A multitude of tests were performed using different number of lights and enabling or disabling shadow generation. All lights used during the tests are spotlights and the shadow maps, while enabled, utilized 1024 by 1024 resolution and were either enabled or disabled for all lights at the same time. The Table 4.2, Table 4.3, Table 4.4 and Table 4.5 show the statistics acquired during these tests. While the higher FPS values shown in these tables may seem out of context and unnecessary when compared to a targeted sustained 60 FPS, they actually serve to show the extra amount of time gained per frame which does not have to be spent rendering. The extra time can be used by other parts of the application such as physics simulations.

Since refraction is simulated in the final composition buffer in the proposed method, the relevant calculations are independent from the number of lights. Therefore the measurement with no lights give the most accurate performance cost, which is approximately 11%. The results in Table 4.2 show that performance impact of simulating refraction dissipates as the number of lights are increased. The shader analysis performed with PIX for Windows [25] showed that the shader with no refraction used 4 texture and 14 arithmetic instructions, while the shader with refraction used 9 texture and 27 arithmetic instructions.

Table 4.3 shows that the extra color buffer included in the shadow map to create transparent shadows decreases the performance steadily with each new light. This is to be expected since any new light would also cause a new color buffer write operation and deinterlacing. The performance cost of transparent shadows goes as high as 34% at 64 lights, and the lowest cost measured is at 2 lights with 13%. The fragment shader which accumulates lighting for shadows was also run through analysis with PIX for Windows [25]. This showed that the shader with opaque shadows used 4 texture instruction slots and 66 arithmetic instruction slots, while the shader with transparent shadows used 7 texture instruction slots and 82 arithmetic instruction slots.

It can be seen in both Table 4.4 and Table 4.5 that the deferred rendering algorithm proposed in Chapter 3 outperforms all of the other methods tested against in terms of speed, starting with the 8 lights mark. The forward renderers could only perform faster when 4 or less lights were used, at a maximum of 33% at 2 lights when compared to the proposed method.

When shadows are enabled, the proposed method is at most 36% faster than the forward renderers at 32 lights. It should be noted that at 64 lights, since the shadow map generation dominates the rendering interval, the speed up of the proposed method against the forward renderer decreases to 28%. When shadows are disabled, when only refraction is enabled, the proposed method is at least 27% and at most 74% faster then the forward renderers starting with 8 lights.

Table4.4: Average FPS during test course with shadows

# of Lights	Proposed Method	Deferred Peeling	Forward Rendering (w depth)	Forward Rendering (w/o depth)
2	318 FPS	197 FPS	390 FPS	**423 FPS**
4	240 FPS	154 FPS	255 FPS	**268 FPS**
8	**166 FPS**	111 FPS	154 FPS	156 FPS
16	**111 FPS**	80 FPS	83 FPS	84 FPS
32	**69 FPS**	53 FPS	44 FPS	44 FPS
64	**32 FPS**	26 FPS	23 FPS	23 FPS

Table4.5: Average FPS during test course without shadows

# of Lights	Proposed Method	Deferred Peeling	Forward Rendering (w depth)	Forward Rendering (w/o depth)
2	441 FPS	239 FPS	526 FPS	**588 FPS**
4	398 FPS	213 FPS	375 FPS	**406 FPS**
8	**337 FPS**	178 FPS	234 FPS	246 FPS
16	**308 FPS**	162 FPS	132 FPS	135 FPS
32	**266 FPS**	138 FPS	70 FPS	72 FPS
64	**143 FPS**	71 FPS	37 FPS	37 FPS

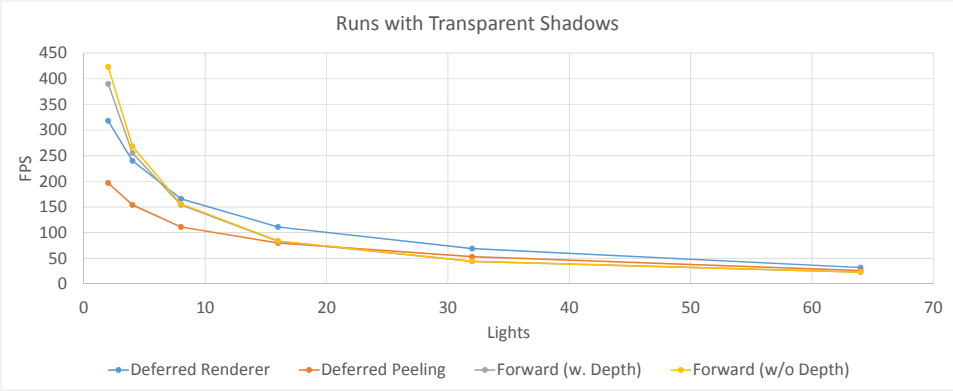

Figure 4.10: Chart of Table 4.4

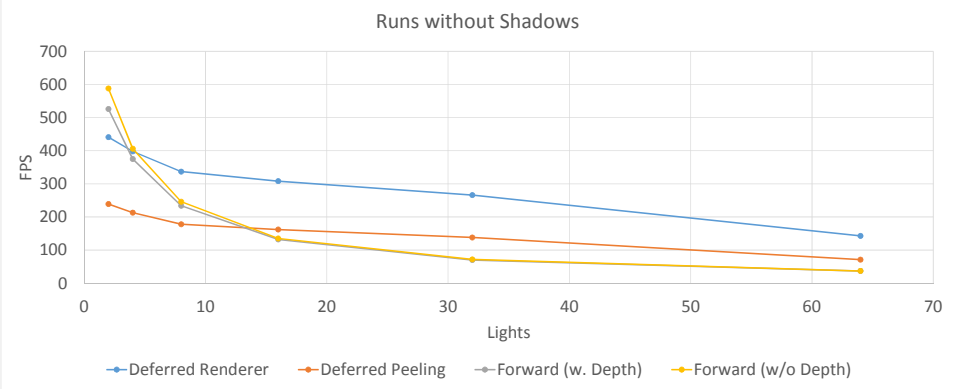

Figure 4.11: Chart of Table 4.5

The final image result of the deferred renderer with depth peeling can be considered the most visually accurate since there is no loss of quality in any part of the image. However it is consistently slower when compared to the proposed method in Chapter 3. When shadows are enabled the deferred renderer with depth peeling is at least 18% slower, where the shadow map passes saturate the rendering interval, and at most 50% slower, when shadows are disabled and both deferred renderers achieve peak performance where the utilization of G-Buffer is justifiable. While it is slower then the forward renderers in some situations, it still manages to flourish in a multiple light situation, starting with 32 lights or more when shadows are enabled and with 16 lights or more when shadows are disabled.

4.3 Performance of Deferred Renderer with Different Amounts of Transparency

To observe the performance of the deferred renderer proposed here with scenes containing different levels of transparency, two additional scenes were prepared containing the same total number of polygons with the initial test scene used in Section 4.2 but different numbers of transparent polygons. Below are the numbers of polygons for each scene. While Scene 1 is the scene used in Section 4.2, the additional scenes are named Scene 2 and Scene 3, respectively.

Scene 1: 38812 opaque polygons + 20832 transparent polygons

Scene 2: 49210 opaque polygons + 10434 transparent polygons

Scene 3: 59644 opaque polygons + 0 transparent polygons

The Table 4.6 and Table 4.7 show the average FPS of the scenes with shadows and without shadows respectively. The maximum FPS difference between the scenes is 4 FPS and the differences do not show any consistency or relation with any of the scenes. ±2 FPS has been observed between the previous runs with the exact same setup and conditions, therefore these FPS differences can be considered negligible fluctuations. These results are to be expected since the deferred renderer proposed here does not differentiate between opacity and transparency at an object level and it does not use different rendering models for them. Since in the current implementation of GPU programs conditional branching does not always prevent unnecessary computations [10], the same amount of computations may occur for both opaque and transparent fragments. In conclusion, these tables show that the performance of the deferred renderer proposed here is not affected by the transparency ratio of a scene when the total number of polygons is constant.

Table4.6: Performance in different scenes during test course with Shadows

# of Lights	Scene1	Scene2	Scene3
2	318 FPS	319 FPS	320 FPS
4	240 FPS	241 FPS	241 FPS
8	166 FPS	166 FPS	166 FPS
16	111 FPS	111 FPS	111 FPS
32	69 FPS	70 FPS	70 FPS
64	32 FPS	34 FPS	34 FPS

4.4 Performance of Deferred Renderer with Different GPUs

The scene used in Section 4.2 was also run with different test rigs, to measure the performance of the proposed deferred renderer with different GPUs. In the

Table4.7: Performance in different scenes during test course without Shadows

# of Lights	Scene1	Scene2	Scene3
2	441 FPS	443 FPS	444 FPS
4	398 FPS	400 FPS	401 FPS
8	337 FPS	338 FPS	339 FPS
16	308 FPS	310 FPS	309 FPS
32	266 FPS	265 FPS	268 FPS
64	143 FPS	142 FPS	139 FPS

descriptions below, the test rig used in Section 4.2 is labeled Setup 1, while the two other additional setups are labeled Setup 2 and Setup 3 respectively.

Setup 1: i7 4770k CPU, 8GB RAM, Geforce GTX 760 2GB

Setup 2: i7 4770 CPU, 16GB RAM, Geforce GTX 690 4GB (2GB + 2GB)

Setup 3: i7 4790 CPU, 8GB RAM, Geforce GTX 780 3GB

Table4.8: Performance of GPUs during test course with Shadows

# of Lights	Setup 1	Setup 2	Setup 3
2	318 FPS	367 FPS	444 FPS
4	240 FPS	265 FPS	332 FPS
8	166 FPS	183 FPS	225 FPS
16	111 FPS	122 FPS	149 FPS
32	69 FPS	76 FPS	90 FPS
64	32 FPS	37 FPS	46 FPS

Table4.9: Performance of GPUs during test course without Shadows

# of Lights	Setup 1	Setup 2	Setup 3
2	441 FPS	506 FPS	624 FPS
4	398 FPS	438 FPS	571 FPS
8	337 FPS	365 FPS	511 FPS
16	308 FPS	324 FPS	473 FPS
32	266 FPS	279 FPS	421 FPS
64	143 FPS	147 FPS	239 FPS

From the results of runs with and without shadows, shown respectively in Table 4.8 and Table 4.9, it can be seen that the GPUs' response to increasing number lights are very similar. As the number of lights increase, each of the GPUs start

to get saturated and as the GPUs become saturated, the performance decrease at each light number step approaches to being linearly inversely proportional to the increase in the number of lights.

CHAPTER 5

CONCLUSION AND FUTURE WORKS

The methods proposed in this study integrate a single layer of transparency into a deferred renderer by simulating the refraction of transparent objects and rendering a single layer of colored transparent shadows, which can appear on both transparent objects and opaque objects alike. Real-time interactive frame rates are maintained while the visual fidelity of the scene is increased without using any additional rendering pipelines.

None of the methods proposed in this study require additional passes over the scene geometry or additional post processing passes when compared to a generic deferred renderer with shadow mapping, as such no additional draw calls are introduced. No kind of spatial partitioning, object sorting or object based drawing methods are required for the methods proposed to work accurately. Any and all calculations are done using programmable shaders.

The evaluations performed up to 64 lights has revealed that neither the refraction simulation, nor the modification for transparent shadows caused the renderer to drop below real-time interactive frame rates, when compared to a default deferred renderer without the effects proposed here.

In a scene with approximately 60000 polygons the cost of simulating refraction in the deferred renderer was observed to be 11%. It should be noted that simulating refraction in a deferred renderer is independent of scene geometry and the number of lights since it is a post-process effect that does not require any additional setup when used in a deferred renderer. Therefore, as scene complexity and number of lights increase, the cost of simulating refraction will dissipate into other increasing costs. The instruction cost of simulating refraction was observed to be 4 extra texture instruction slots and 13 extra arithmetic instruction slots.

When compared to opaque shadows, the cost of creating transparent shadows for a scene with 60000 polygons was observed to be a maximum of 34% at 64 lights.

When compared with forward renderers implementing the same visual effects in similar ways, the proposed method was at least 6% and at most 28% faster than the forward renderers when the number of shadow casting lights used was more than 4. This is to be expected of any deferred and forward renderer comparison.

The proposed method was faster than the deferred renderer with depth peeling in every condition, a minimum speed up of 18% was observed at 64 shadow casting lights at the cost of decreased resolution in the areas of the image with transparent objects.

Three main contributions are presented in this study. Firstly, a new implementation for transparent shadows for a single layer of transparent occluders was provided and it was shown to be applicable to multiple rendering pipelines. Secondly, previously proposed refraction simulation was shown to be applicable to deferred rendering, given that it supports at least a single layer of transparency. Finally, an improvement upon a previously proposed refraction simulation was presented. This improvement corrected some rendering errors that could manifest as incorrect refraction results, while implementing this improvement had no additional cost in the deferred renderer it came at the cost of creating a depth map of the scene in the forward renderer.

5.1 Future Works

While the methods proposed here have revealed to maintain real-time interactive frame rates with their current implementations, the shadows created and objects appear aliased. For these techniques to be applicable to games, they need to be improved with anti-aliasing and softer shadows.

The synergy between the methods proposed here and other visual effects, such as ambient occlusion or HDR, should be investigated. Also the evaluations were done in three scenes with the same number of polygons, without any optimization done on the CPU side. The real applications of these methods will only be possible after performing optimization such as including a spatial partitioning techniques to decrease the number of objects drawn on the shadow maps.

Point and directional light shadows have also not been implemented or evaluated, while both are frequently used in virtual reality applications.

Another avenue to follow would be to perform user experience testing on a game using the deferred renderer proposed here. This user experience evaluation can be used to find the significance of the quality decrease caused by interlacing the transparent surfaces and the possible increase in experience through the methods proposed here.

REFERENCES

[1] S. Brabec, T. Annen, and H.-P. Seidel. Shadow mapping for hemispherical and omnidirectional light sources. In *Advances in Modelling, Animation and Rendering*, pages 397–407. Springer, 2002.

[2] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt. The triangle processor and normal vector shader: A vlsi system for high performance graphics. *SIGGRAPH Comput. Graph.*, 22(4):21–30, June 1988.

[3] Deferred Rendering Tutorial by Catalin Zima. `http://www.catalinzima.com/xna/tutorials/deferred-rendering-in-xna/`. [30-July-2014].

[4] Directly Mapping Texels to Pixels (Direct3D 9). `https://msdn.microsoft.com/en-us/library/bb219690.aspx`. [15-February-2015].

[5] C. Everitt. Interactive order-independent transparency. *White paper, nVIDIA*, 2(6):7, 2001.

[6] R. Fernando. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[7] D. Filion and R. McNaughton. StarCraft II effects & techniques. In *ACM SIGGRAPH 2008 Games*, SIGGRAPH '08, pages 133–164, New York, NY, USA, 2008. ACM.

[8] H. Gouraud. Continuous shading of curved surfaces. *Computers, IEEE Transactions on*, C-20(6):623–629, June 1971.

[9] S. Hargreaves. Deferred shading. Game Developers Conference. 2004. `http://developer.amd.com/wordpress/media/2012/10/D3DTutorial_DeferredShading.pdf`. [30-July-2014].

[10] M. Harris and I. Buck. Gpu flow control idioms. *GPU gems*, 2:547–555, 2005.

[11] J. Klint. Deferred rendering in leadwerks engine. Copyright Leadwerks Corporation ©2008. `http://www.leadwerks.com/files/Deferred_Rendering_in_Leadwerks_Engine.pdf`. [30-July-2014].

[12] R. Koonce. Deferred Shading in Tabula Rasa. In *GPU Gems 3*, volume 3, pages 429–457. Addison-Wesley Professional, 2007.

[13] M. Mittring. A bit more deferred–CryEngine 3. Triangle Game Conference. 2009. www.crytek.com/download/A_bit_more_deferred_-_CryEngine3.ppt. [30-July-2014].

[14] Multiple Render Targets (Direct3D 9). http://msdn.microsoft.com/en-us/library/windows/desktop/bb147221(v=vs.85).aspx. [30-July-2014].

[15] D. Pangerl. Deferred rendering transparency. In W. Engel et al., editors, *ShaderX7: Advanced Rendering Techniques*, ShaderX series, chapter Rendering Techniques, pages 217–224. Course Technology/Cengage Learning, 2 edition, 2009.

[16] B. T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[17] T. Porter and T. Duff. Compositing digital images. *SIGGRAPH Comput. Graph.*, 18(3):253–259, Jan. 1984.

[18] W. T. Reeves, D. H. Salesin, and R. L. Cook. Rendering antialiased shadows with depth maps. *SIGGRAPH Comput. Graph.*, 21(4):283–291, Aug. 1987.

[19] T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 197–206, New York, NY, USA, 1990. ACM.

[20] O. Shishkovtsov. Deferred shading in S.T.A.L.K.E.R. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*, chapter 9, pages 143–165. Addison-Wesley Professional, 2005.

[21] T. Sousa. Generic refraction simulation. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*, chapter 19, pages 295–306. Addison-Wesley Professional, 2005.

[22] H. Stroyan. Fast alpha transparency rendering method, July 13 1999. US Patent 5,923,333.

[23] Unity Deferred Lighting Rendering Path. http://docs.unity3d.com/Manual/RenderTech-DeferredLighting.html. [30-July-2014].

[24] Unreal Engine 4 Documentation - Rendering Overview. https://docs.unrealengine.com/latest/INT/Engine/Rendering/Overview/index.html. [30-July-2014].

[25] Where is the DirectX SDK? https://msdn.microsoft.com/en-us/library/ee663275(v=vs.85).aspx. [15-February-2015].

[26] L. Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, Aug. 1978.

[27] XNA Game Studio. `https://msdn.microsoft.com/en-us/library/bb200104.aspx`. [15-February-2015].