University of Dublin, Trinity College

Masters Thesis

Optimization to Deferred Shading Pipeline

Author:
Samir Kumar

Supervisor:
Dr. Michael Manzke

August 2017

A thesis submitted in fulfillment of the requirements for the degree of MSc Computer Science (Interactive Entertainment Technology)

in the

IET
School of Computer Science and Statistics

Declaration of Authorship

I, Samir KUMAR, declare that this thesis titled, "Optimization to Deferred Shading Pipeline" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at this University.
- I declare that this thesis has not been submitted at any other university.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Date:

_____

University of Dublin, Trinity College

Abstract

The Faculty of Engineering, Mathematics and Science

School of Computer Science and Statistics

MSc Computer Science (Interactive Entertainment Technology)

Optimization to Deferred Shading Pipeline

by Samir KUMAR

Rendering pipelines in the field of graphics are continuously evolved. Deferred rendering pipeline is a step forward from most Common pipeline Forward rendering pipeline. Deferred rendering pipeline has its disadvantages such as its does not support blending. The aim of this project is to optimize the deferred rendering pipeline to increase its usability for a large variety of project. One of the objectives of the project is to solve the problem of blending by integrating forward pass as the third pass into the deferred pipeline to render blending. Next, comes the Light volumes calculations to calculate lighting and increase computational efficiency. After couple experiments and dealing with results of these experiments, blending is successfully integrated. It works efficiently and justifies deferred characteristics of rendering a large number of objects and lights. The results produced by blending implementation are realistic and efficient. Lights rendered to perform blending uses the distinctive shader. Light volume calculation approach is also successfully implemented. However rendering performance improves but it does not hit the expected mark. The reason behind these results is due to GLSL and GPU characteristics. However, this can also be solved by the approach suggested in future work. This optimization of the deferred pipeline can be utilized to render a large number of objects and light while allowing the use of blending and special shader, which is something not possible in generic deferred shading. This approach also supports customizations and scalability depending on requirements of the project.

Acknowledgements

I would first like to thank my thesis advisor Dr. Michael Manzke of the School of Computer Science and Statistics at Trinity College Dublin. The door to Prof. Manzke office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work but steered me in the right the direction whenever he thought I needed it.

I would also like to acknowledge Dr.John Dingliana of the School of Computer Science and Statistics at Trinity College Dublin as the second reader of this thesis, and I am gratefully indebted to him for his valuable comments on this thesis.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you.

Author

Samir Kumar

Contents

Declaration of Authorship **iii**

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**
1.1 Motivation 2
1.2 Objective 3

2 Background **5**
2.1 Research Question 6
2.2 State of the Art 8
2.2.1 Tile Shading 8
2.2.2 Alpha Blending 9
2.2.3 Clustered Deferred Shading 11
2.2.4 Light Volumes 12
2.2.5 Bling Phong Lighting 13
2.3 Reflection 15

3 Implementation **17**
3.1 Experiment 17
3.2 Requirement 18
3.2.1 Framework 19
3.3 Custom Deferred Implementation 19
3.3.1 Geometry Pass 19
3.3.2 G-Buffer 21
3.3.3 Initial Algorithm 22
3.3.4 Lighting Pass 24
3.3.5 Limitation of this implementation 26
3.4 Forward Pass integration into Deferred Pipeline 26
3.4.1 Algorithm Improvisation 1 27

3.4.2 Interaction between existing Deferred Renderer and Forward pass . . . 29
3.4.3 Algorithm Improvisation 2 . . . 29
3.5 Light Volumes . . . 32

4 Evaluation and Results **37**
4.1 Evaluation . . . 37
4.1.1 Geometry Pass textures . . . 37
4.1.2 Performance Analysis . . . 39
GPU and CPU utilization variation test . . . 40
4.1.3 Light Volumes Vs No Light Volumes . . . 41
Without Light Volumes . . . 42
With Light Volumes . . . 42
4.2 Limitations . . . 44

5 Conclusion **45**
5.1 Conclusion . . . 45
5.2 Future Work . . . 46

A Code Listing for Rendering all colorbuffers in to G-Buffer **47**

B Code Listing Of Frame Buffer directly sampled from G-Buffer **49**

C Code Listing Updated Lighting pass fragment shader **51**

D Code and Document Link **53**

Bibliography **55**

List of Figures

2.1 Performance results of Tile Deferred Shading with general deferred shading and other algorithms (Olsson and Assarsson 2011a) 9
2.2 Graph with performance results of Alpha Blending with deferred shading (Magnerfelt 2012b) 10
2.3 Graph showing results of Attenuation equation inspired from (Jaud, Anne-CÃ©cile Dragon, et al. 2012) 13
2.4 Angles demonstration of Blinn Phong Lighting Model 14
2.5 This is the figure shows the difference between Phong and Blinn Phong Lighting Model 15

3.1 Visual of textures stored in the G-buffer (Oosten 2015) 20
3.2 Result of implementation of textures stored in the G-buffer. . . 24
3.3 Result of implementation deferred rendering so far without blending 26
3.4 First experiment to integrate forward pass to deferred pipeline 28
3.5 Deferred rendering and blending with right depths. 32
3.6 Deferred rendering and blending using Light Volumes. 35

4.1 CIE RGB color space diagram in 2D and 3D (Frich 2017) 38
4.2 G-Buffer frame buffer textures colors and normal. 38
4.3 Graph of GPU and CPU utilization without Light Volumes. . . 42
4.4 Graph of GPU and CPU utilization after Light Volumes. 43

List of Tables

4.1 Performance Analysis table for different scale of objects and lights. 39
4.2 Table of CPU and GPU Utilization For Variation of Object and Light Number 41
4.3 Table of performance comparison between With and Without Light Volumes 43

List of Abbreviations

FSAA	**F**ull **S**creen **A**nti **A**liasing
FPS	**F**rame **P**er **S**econd
FT	**F**rame **T**ime
CPU	**C**entral **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
GLSL	**G**raphic **L**ibrary **S**hading **L**anguage

Physical Constants

Attenuation Constant $K_c = 1$

List of Symbols

c	constant	
l	linear	
q	quadratic	
d	distance	cm
r	radius	cm

Chapter 1

Introduction

Over the last numerous decades, the significant need for 3d graphics has been noticed. Due to raising requirements, there have been vast improvements in rendering of graphics and graphic pipelines. Graphics rendering is moving towards realism and getting faster. From very basic imagery to high-end graphics, lighting is the most crucial factor. There are many different ways to render lighting in the world of 3D graphics. One of the standard and straightforward approach is forward rendering. Forward rendering let down when the number of lights is significant. For a large number of lights, the more efficient technique used is deferred rendering. Not only it allows to process a large number of lights but also lifts the scene up towards realism.

Deferred rendering used to render the lighting where forward rendering would break. Deferred rendering has its own limitations. A common flaw is that it does not handle transparency or blending. All the values of g-buffer are from single fragments where as to render blending combination of multiple fragments is required. This aspect has been the matter of research for long, but there is still no ideal method. It is debatable that methods available in industry to alleviate this limitation are worth the project or not considering the efforts they require in implementation.

Some of the techniques that handle the limitation of blending include alpha blending by (Magnerfelt 2012a), Clustered Shading (Olsson, Billeter, and Assarsson 2012a) and more. More detail on these methods is mentioned in the background research section. The industries have been working on the solutions for this limitation such as Climax (Hargreaves 2010), Science Direct using deferred blending (Zhang and Pajarola 2007). Famous game developers have been researching techniques such as Antialiasing by Nvidia. (Chajdas, McGuire, and Luebke 2011). Having all these approaches with their issues and limitation makes them suitable for one and unsuitable for another scenario.

There is still vast room for the simplistic and optimized approach that can handle a large variety of scenes.

1.1 Motivation

Improvements and advancement in the graphic processor units with time made the drawbacks of forward rendering technique easily noticeable. As afore mentioned deferred rendering techniques is utilized to overcome these disadvantage in modern day rendering techniques. This method drastically changes the way we render the scene and the object in it with respect to lighting. (Thaler and Wien 2010) Deferred rendering is based on the core idea of postponing the most expensive parts of rendering such as lighting to the later stage of the rendering pipeline. Resulting in a separation of rendering pipeline into two main part Geometry stage and lighting stage.

(Policarpo, Fonseca, and Games 2005) one of the main disadvantage of this technique, this project is trying to alleviate is that it does not support rendering of lesser opaque objects. So blending or transparency has been a major issue of deferred shading in the graphics industry. The approaches present in the field of graphics industry to overcome this problem are, such as using alpha blending by (Magnerfelt 2012a) and tile shading (Olsson and Assarsson 2011b). Clustered Shading (Olsson, Billeter, and Assarsson 2012a) is another extensive approach. These techniques as also mentioned before do solve the problem of blending but they have their compromises. These trade-offs are detailed in the background research section of the report. Looking into these approaches, one thing that collectively comes out is that they have their specific fields of application and they are hard to adapt to different scenarios. The reasons behind less feasibility for adaptation are their development time input and particular limitations. Mostly they will not be feasible for small scale projects. This collective limitation brings us to the room for a first go to approach. An approach that can solve the limitation of blending from deferred shading, an approach that can be extended and scaled according to the needs of development. These factors become a motivation to develop a suitable approach in this project that will fulfill these requirements.

1.2 Objective

The approach implemented in this project is to deal with this limitation of blending in deferred shading. This project is based on optimizing deferred pipeline to render blending and a large number of lights efficiently. To make blending work with deferred shading forward rendering is brought back. It is utilized as an internal part of deferred rendering pipeline after the shading pass. This integration is done in such a way that the forward rendering only handles the blending part of the scene while being part of the deferred rendering pipeline. This way pipeline will provide the advantages of deferred shading such as rendering high number lights without getting slow as well as allow the use of special shaders and perform blending which is not allowed in the generic deferred rendering pipeline.

This project also consists of work to optimize the pipeline to render a large number of lights using light volume calculations as the generic pipeline will not do it on its own. Using light volumes increases efficiency and speed of the rendering as only the effective area of each light will be calculated for lighting in the scene instead of globally calculating the whole scene for each light source.

As per road map of the project, first deferred rendering pipeline will be implemented. This will be the generic deferred rendering pipeline but custom built to get the full functionality for any future tweaks or addition that are going to be made. Reporting the limitation and disadvantages of this pipeline, a principal aim of the approach is providing the solution to them. Simple forward rendered light sources emitting color will not be sufficient to perform blending. The objective is to get the best realistic results. Then the Forward rendering pass will be integrated into to the existing deferred pipeline such that they exchange the depth values of the objects.

Deferred rendering is praised for its capability to rendered large number of lights. Next part of the project is focused on this element as generic deferred rendering does not do this on its own. To make this possible light volume or radius calculations will be used. These volumes will define the area where light can reach. Finding out this area of each light is the vital part of the solution. In the end, a light will only calculate lighting for the area that resides in its volume.

Chapter 2

Background

Until a couple of years ago, forwards rendering has been the streamlined approach for rendering real time engines. In this method, geometry is sent through the vertex shader. The pixels that are produced are then shaded in the different pass also called fragment or shader pass. Most scenes have the list of object to be shaded with special effects and lighting. (Thaler and Wien 2010) Most expensive part of shading is lighting. It can usually be done in two different ways. First one is single pass and second is multiple pass. There are many drawbacks of this approach which can easily be noticed such as complexity in calculating and rendering light and each material in the scene need personalized, specific shader for every light type.

To address these drawback and performance issues deferred rendering techniques is utilized. Although this technique is not new, it was dated back in 1988, but it was not called deferred rendering. It was introduced by (Deering et al. 1988). In detail information on the implementation of deferred shading can be found on in the publication by (Policarpo, Fonseca, and Games 2005).

This technique is well known and used by many gaming industries and developers. Where it helps a lot with the efficient rendering of a vast number of lights, its disadvantage of not supporting blending or transparency is a major setback. As aforementioned, this is the problem that this project is aiming to solve. There are other techniques present in the industry to address this issue. Following is research on these techniques and how they affect the rendering.

2.1 Research Question

To solve the obstacle of blending it is necessary to discuss the fundamental reason of why this problem originates. It has been debated several times in the past. So here is the brief explanation of what is happening when deferred shading is put into implementation. The following research will help to recognize better what leads to the inability of transparency and blending. As afore mentioned deferred shading is based on the key intent of decoupling of geometry calculations and lighting calculations.

First pass also known as geometry pass is responsible for rendering a screen space representation of the scene that is going to be displayed. This pass is later accumulated in different buffers. Having all the information stored in these buffers, they are utilized in the calculation of lighting pass. Geometry stage renders the attributes of geometry and its materials. There are at least four main attributes that are commonly present in all the deferred pipeline for geometry pass. These four attributes are Normals, Diffuse Color, Specular Color and Shininess, and Depth. G-buffer is split into multiple attributes as it is not possible to store everything in one buffer. To develop geometry pass this way, Deferred Rendering uses MRTs. Deferred rendering requires simultaneous rendering of 4 back buffers which is allowed by MRT. MRT has a serious restriction. It only allows these four buffers if they have equal depth. It uses 32-bit depth. There are several ways to reduce the number of channels needed.

The second stage is Lighting stage. Geometry pass gives the access to four textures. These four textures store above mentioned four attributes. These four textures are used by the lighting pass to calculate lighting for the scene. All the lights those affects the objects in the scene are emphasized, and lighting equation is applied on to these additively. Lighting equation is based on the attributes of geometry pass.

Further, lies the implementation of light volumes to make the light calculations more efficient. There is a various number of light volume calculation models that can be used in the implementation. The implementation depends on these light volume calculations to increase efficiency. Both pixel and vertex shader is active at this stage. Pixel shader does not produce the same output as it does in forward rendering. Pixel shader differs as the pipeline is deferred rendering pipeline.

Geometry and lighting stages are two most crucial stages of deferred shading. Following these two steps, implementation proceeds to the post processing stage. This stage produces effects like Bloom, HDR, Motion Blur, and so forth. It uses the output from lighting pass, and it also requires information from geometry stage to produce these effects. The final stage is displaying these results on the screen. It is done by utilizing the quad polygons and texturing them. Quad polygons have the size of screen resolution.

A visibility test is used in the first pass of the approach, to expedite and prevent the rendering of obstructed geometry. This test underwrites that only lighting calculations those are important for the final image are inserted. The problem that occurs here is that only pixel worth information is stored at the time of visibility test due to the limitations of the graphics hardware. This information is not sufficient to perform blending. In deferred shading, all the G-buffer Values (first pass) are also stored in the single fragment. To perform blending in the scene it requires multiple fragments. These issues also lead to the challenge of rendering the transparent objects or blending in the scene rendered by deferred shading. To directly render transparency into the deferred pipeline is not possible because one first hit with the surface of the object is recorded. Multi layer frame buffer does not exist as proposed by (McGuire and Bavoil 2013).

Generic deferred shading pipeline also does not help in rendering a lot of light numbers without heavy cost of performance. The general form deferred pipeline has to calculate the lighting of each fragment for every light source present in the scene. In other words, there is a heavy amount of calculations for the lights that need to be done. These calculations are most of the time not even necessary for e.g. a light lamp in the bed room. This lamp does not light up the whole room. However, to create a light lamp in the bed room scene, one has to calculate the entire room for that light source. These calculation include the fragments where light does not reach. So for two light lamps, the calculations get multiplied and so on for three and more lights. Having a large number of lights will then put an enormous amount of load on the performance most of which is the waste of lighting computations. So this is another problem of deferred shading that will be dealt with in this project's implementation using light volumes.

2.2 State of the Art

2.2.1 Tile Shading

State of the art solutions provided by different researchers and industries are introduced in the previous sections of the report. Tile shading is one of the well-known technique used to enhance the deferred rendering pipeline. It has been discussed multiple times (Balestra and Engstad 2008; Andersson 2009; Swoboda 2009; Lauritzen 2010). It focuses on transparency and blending that we are trying to solve in this project by implementing rendering differently. It was developed to advance both deferred and forward rendering (Olsson and Assarsson 2011b). It is very similar to Tiled rendering (Fuchs et al. 1989). The way they made it work was by bucketing up the lights into the screen space tiles such that each tile consists of some potential light sources. In this technique, the geometry with transparent properties is rendered into the g-buffer. The frame buffer is then incorporated by constructing the screen space grid. This grid has fixed tile sizes such that $f = (x, y)$ where f = 32 x 32 pixels. Next screen space is calculated for each light up to the extent of its light volume. Resulted grid cells were appended with light ID that needs to be applied. Each fragment that is in the frame buffer with location $f = (x, y)$, g-buffer is sampled at f. Contribution of light from all light sources is collected in the tile at f/t. Finally, all the light output is forwarded to frame buffer at f.

To evaluate the performance results of this technique PC with Intel quad core two at 2.5 GHz was used. The graphic cards used to measure the results were either NVIDIA GTX 280 or GTX 480 GPU. Rendering was done at full HD 1920x1080p. Among all test scenarios, there was *General deferred shading*, *Deferred Stencil* with stencil optimization and *Tile Deferred* using CUDA to build grid and also depth range and lighting computations. Performance graph is next shown with all the variable techniques in testing.

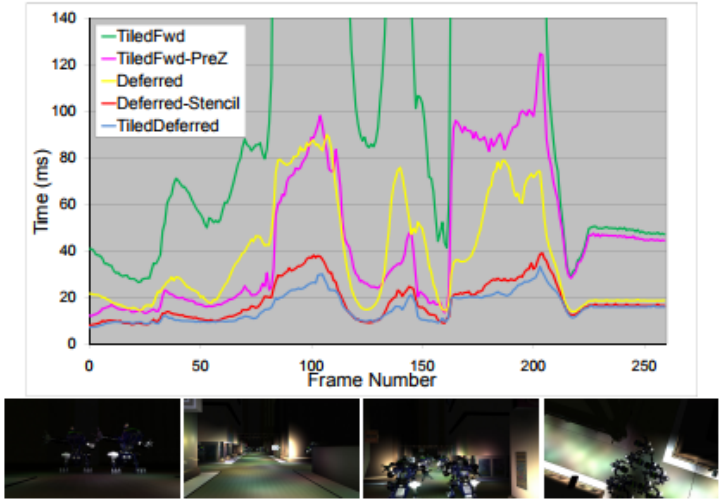

FIGURE 2.1: Performance results of Tile Deferred Shading with general deferred shading and other algorithms (Olsson and Assarsson 2011a)

Tile shading performance can be examined from the graph in figure 2.1.(Olsson, Billeter, and Assarsson 2012b). It states that performance of tile shading does not vary much at best case or worst case scenarios. Other experiments on this approach suggest that frame time does linearly increase with the increase in G-buffer size. The hardware used for the performance test is with high specifications. To get best results from this approach hardware cost is likely to be high.

The weaknesses of generally deferred shading are also shared by the tiles shading method. The main issue from this techniques is the requirement of the frame rate when used with *FSAA*. Using general 1080p with 16 times *MSAA* and a 32-bit color needs 256Mb to handle the storage of depth and color samples. It is impossible for most of the current hardware in the industry as there will be several G-buffers added on top of that. This addition on the top will increase the memory requirement immediately. Transparency is only done by approximation (Kircher and Lawrance 2009) as it does not have a full solution in tiled rendering technique.

2.2.2 Alpha Blending

Another approach is of alpha blending (Magnerfelt 2012a). Developers of this method used the alpha blending as post pass using the front rendering. They consider it to be straight forwards to implement into the deferred rendering. However, one has to possess the knowledge of flaws of alpha blending. The

algorithm in this approach is based on using an alpha channel to store transparency. Opacity data is stored in this channel. Opacity levels are from 0.0 to 1.0 which decides how transparent the object is going to be. RGBA is used to represent the alpha value. Here A determines the final color of the object. After calculating this value, it is then stored into the frame buffer. Alpha value A is multiplied to rest of the RGB. Previous rendered fragments color is then also multiplied with 1 alpha of currently rendered fragment. So, if the current fragment is made of 0.75 alpha value, then the previous will only be visible 0.25. The above blending method is accessed by the OpenGL functions.

Implementation of this technique lies in 5 steps. Firstly rendering all the non-transparent objects to the G-buffer and then binding the G-buffer as one texture. A rectangle is drawn on the screen for each light. Data stored in G-buffer is used to do the lighting calculation for each pixel. The result is then stored in P-buffer. Sorting as mentioned earlier from back to front order for all transparent objects. Front renderer then renders all the transparent objects. If any object is fully transparent, it is filtered out by P-buffer using depth values(Magnerfelt 2012a). So, the pipeline can ignore the calculations for the fully transparent object.

All the performance tests are done on Intel Core i7-2600 3.40 GHz CPU. GPU used is Radeon HD 6570 with the operating system of windows 7 64bit. 1280x800 and 800x600 are the two resolutions tested considering modern applications. This approach successfully allows the blending and transparency to a certain extent. Below figure 2.2 shows the performance results of the techniques with Frames per sec in regards to the number of lights.

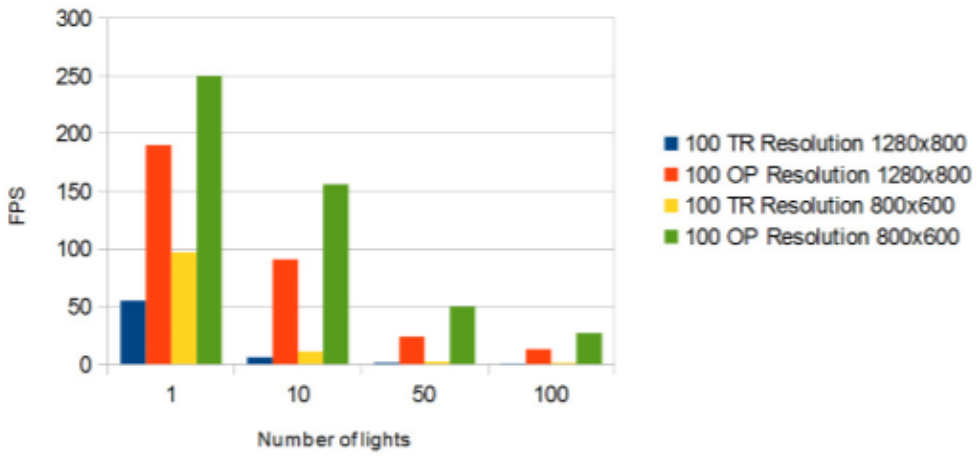

FIGURE 2.2: Graph with performance results of Alpha Blending with deferred shading (Magnerfelt 2012b)

Notice that in the comparison between transparent and opaque objects, transparent objects have a much lower frame rate. Only going up to ten lights has a major effect on the performance for rendering transparency or blending. When the number of lights grows to 50 and 100, the frame rate of rendering is negligible. These test results prove that this technique is inefficient of rendering a large number of lights.

After the experiments, results of this approach shows that with multiple lights, object order affects the scene in a negative manner. (ibid.). Sorting order of objects leads to the poor frame rate. Researcher claims that rendering of each light for all object causes the problem. To avoid this issue, rendering all lights for each object is necessary. It is also essential sort all the polygons in order. This approach would not be feasible with a large number of lights as each light have to rendered for every single object and then blended individually. It results in high computational cost, and its effect can be seen in the test results. Hardware cost would be high to acquire the best results. Performance testing uses even higher specs hardware than *Tile Shading* and still the results are not exceptional. So graphic hardware cost can not be ignored.

2.2.3 Clustered Deferred Shading

Clustered Deferred Shading (Olsson, Billeter, and Assarsson 2012a) is another critical approach to solve the problem of transparency or blending. It is a step forward from tile shading. Higher dimensional tiles which in a collection are called clusters. For each cluster, there is no degenerated case in regards to the view as each cluster is organized with the 3d extent of fixed maximum limit. The main focus is on the approach of clustered deferred shading algorithm as it is more relevant to the topic of research. It is formed of basic five steps. In brief, first one is to render the scene into g-buffers. The second phase is assigning the clusters to the view samples. Integer key is computed to the given view samples in the G-Buffers. Positions and/or normals are used to do this computation. Thirds step is finding the unique clusters. To find the unique clusters approach uses two options, one is sorting, and another one is page tables. Cluster keys are simply sorted, and compaction is performed on it which removes the same neighbors. These methods of sorting and compaction are efficient and are easily available (Billeter, Olsson, and Assarsson 2009; Hoberock J 2010). The fourth step is assigning the light to the clusters. This stage calculates all the light that is influencing each cluster.

This approach introduces the fully favored support for a large number of lights, unlike the tiled shading. This hierarchal approach constructs BVH for each frame by sorting lights in regards to Z order. The last step is shading the samples. Unlike tile shading where simple 2D look up is sufficient this approach differ significantly. Index for each pixel is stored explicitly using the sorting approach.

Performance testing mostly uses NVIDIA GTX 480 GPU as graphic hardware. This method shows how the implementation of clustered shading can be efficient in both deferred and forward rendering. To test out the approach they used various scenes including *Necropolis* and famous *Sponza* model. *Necropolis* (Unreal 2011) scene for unreal development kit consists 654 lights all with bounded range. All lights in the scene are built as point lights. There are 2M triangles, and all of them are normal mapped.*Sponza* Model by Crytek (Crytek 2010) which made the scene more complex and increased discontinuities to better test the performance. There were 10 thousand random lights in the scene.

One of the main advantage of Clustered shading in comparison to tiled shading is lesser view dependency. Still having clustering and light assignment, it causes overhead. So in the case of having a lower light number or lesser discontinuity, tile shading will perform better than clustered shading technique. An amount of efforts put to implement clustered shading approach is considerably high. So developers need to consider the effort in comparison to the resulting outcome of this technique.

2.2.4 Light Volumes

Moving on the light volumes, this is the final proposed optimization to the pipeline after blending. The aim is to reduce the computations for lighting calculations only up to where light travels and matter. Light traveling from the point and slightly reduces as it gets farther from the light source is called attenuation (Markager and Vincent 2000). One way of doing it is using the linear equation. A Linear equation can make the light amount decrease with the increasing distance. Results of using linear equation can be unrealistic. The reason behind having unrealistic results is that in real life the light tends to heavy closer to the light source and then it suddenly falls after a small distance and after that light amount decreases slowly unlike linear equation where the drop in light amount is constant (Cohen 1971). To solve this, the

equation used for light attenuation is

$$F_{light} = \frac{1}{K_c + K_l * d + K_q * d^2}$$

Here d represents distance from the origin of the light source. For the calculation of light attenuation, three values are defined. These three are, first a constant term Kc, second is a linear term Kl, and the last one is a quadratic term Kq. Constant 1 on the right side of the equation is so that the value of denominator does not get lower than 1. When the liner term gets multiplied by the distance, it linearly reduces the intensity of light. The quadratic decrease of the intensity of the light source is done by the quadratic term in the equation. It is multiple with the quadrant of the distance. (Jaud, Anne-Caocile Dragon, et al. 2012) Then it decreases suddenly over a certain distance and after that the drop in the brightness in at plodding pace. Below is the graph 2.3 showing the results of attenuation equation.

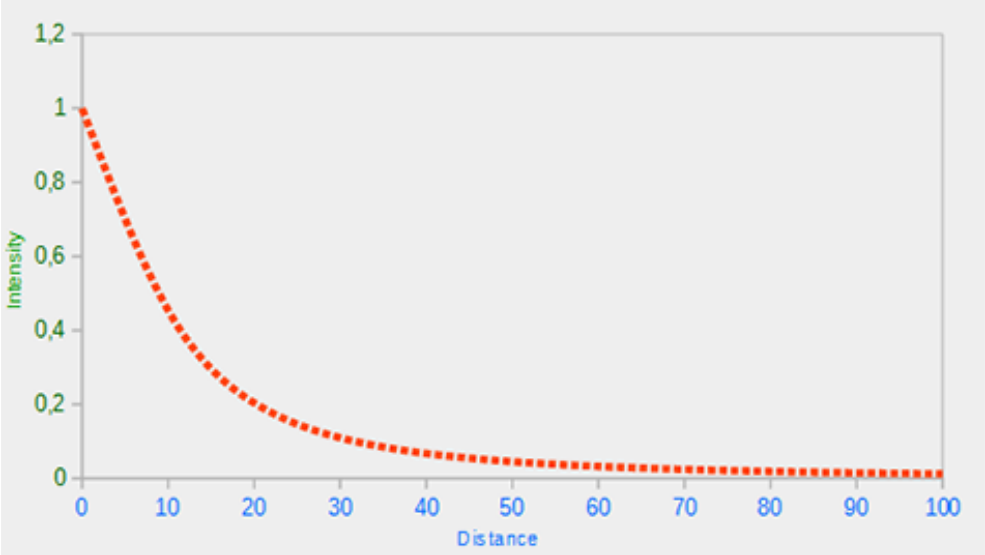

FIGURE 2.3: Graph showing results of Attenuation equation inspired from (Jaud, Anne-CÃ©cile Dragon, et al. 2012)

2.2.5 Bling Phong Lighting

Bling Phong (Blinn 1998) is a step from the Phong Lighting model. Phong (Phong 1975) is very efficient and moderately realistic lighting model among all basic lighting models. Phong lighting has some disadvantages such as specular reflections breakdown especially in areas of low shininess. These areas have the rougher specular breakdown. Specular area cut off the edges of

Phong lighting can be easily seen in the scenarios where reflection and view vector have angle more than 90

(Blinn 1977) introduced Blinn- Phong shading model in 1977. It was introduced as an extension to the basic lighting model Phong. This model had a different approach to specular lighting which eliminated the drawback of light break down on the edges. This model uses a vector called *halfway vector*. This vector replaces the reflection vector used in basic Phong Lighting model. *Halfway* vector is exactly in the middle of view and light direction. So the specular beneficence increase as the halfway vector comes close to the normal vector. Figure 2.4 shows how the angles exist of the surface.

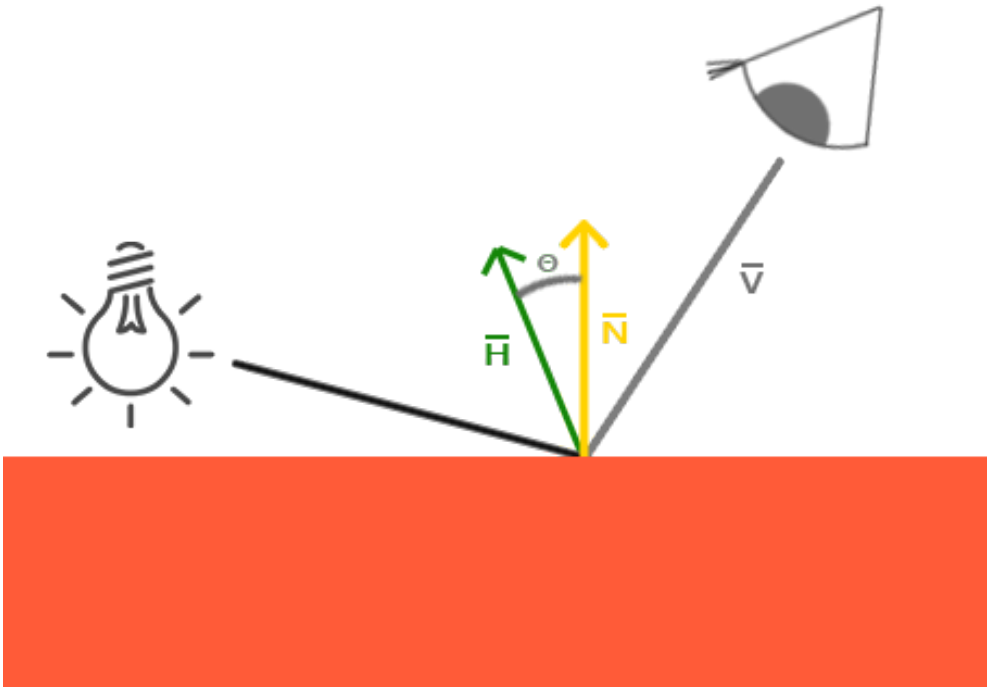

FIGURE 2.4: Angles demonstration of Blinn Phong Lighting Model

Now it is clear that the angle between normal and specular vector will never increase 90 degrees, no matter from where the viewer looks at it. This change outputs more realistic and visually appealing results than Phong Lighting model. Figure 2.5 displays the difference between Phong and Blinn Lighting Model.

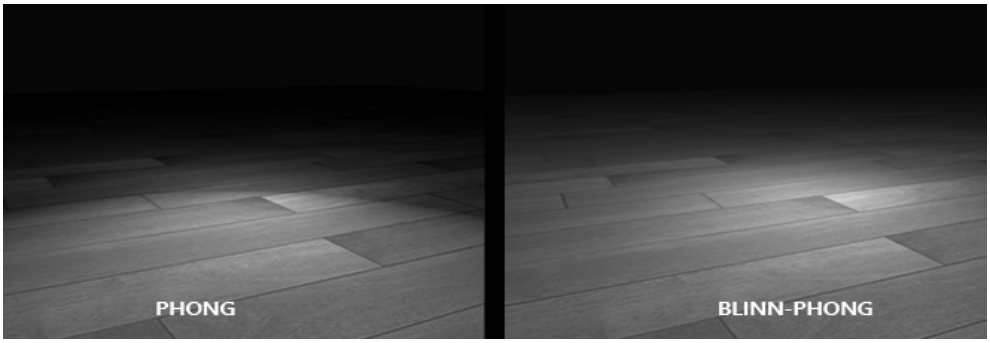

FIGURE 2.5: This is the figure shows the difference between Phong and Blinn Phong Lighting Model

Bling Phong Lighting is the lighting model the approach this project is going to use in the implementation. Blinn Phong Lighting model is the advanced lighting model and produces realistic results with more efficiency.

2.3 Reflection

Previously mentioned state of the art consists a couple of solutions for the problem this project is aiming to solve. Methods referred to in state of the art have their advantages and limitations. Such as Tile Shading share many common drawbacks of deferred shading. The main problem with this approach is that requires very high frame rate when used with FSAA. FSAA is the what makes blending and transparency possible in this technique. Even when used with MSAA, having 1080p requires a very high amount of memory to store the depth and color samples. This requirement is nearly impossible for the modern graphics hardware which makes it less useful for generic or low budget applications. Where as a technique aimed to develop in this project will be far more generic and can be implemented on various hard wares.

Alpha Blending undoubtedly makes blending and transparency possible but at a very high cost of computation. This technique has an issue with the order of object concerning light. After the implementation of this technique, the results found were not accurate (Magnerfelt 2012a). This technique requires rendering of each light or each object along with sorting of all the polygons. It requires very heavy computations, and it is opposite to the aim of the project. Project's approach is to optimize deferred shading to get faster results while supporting transparency and blending. The project aim is at more generic the approach to be used on all various scales. Alpha blending will not be feasible

for a large number of lights, the number of computations is multiplied with an increase in the number of lights. Our optimization technique focuses on reducing the wastage of unnecessary computations by using Light volumes.

Cluster shading is a next step up from tile shading. It is beneficial for both forward and deferred shading. The key element of this approach such as clustering and light assignment cause overhead. In comparison to tile shading having a low light scenario, tile shading performs better than cluster shading. This procedure deals with problems residing in the tile shading. Still, after the huge amount of development put into this technique it is outperformed by tile shading when having lesser number of lights. This technique is highly efficient to render a large number of lights. It is also advantageous in many more scenarios. Implementation of this technique requires a lot of efforts and time into the development, and one has to decide if it is worth implementing for their project or not.

Not only this approach but all the other techniques are also needed to be thought of in regards to development time and efforts that go into them. The amount of time and effort should be worth the outcome. It also depends on the requirements of the project. On the other hand, our approach is far more feasible with the lower number of lights. It is aimed towards a reducing development time while getting good results and higher quality rendering. It will support blending, and our approach can be amended according to the needs of development

Chapter 3

Implementation

After background research of the deferred rendering pipeline, its limitations, and state of the art to deal with those constraints, the objective of the project is evident. As afore mentioned, the aim is to optimize the deferred rendering pipeline by solving the problem of blending and using light volumes to increase the computational efficiency. This section of the report will shed light on the process of implementation of an optimized pipeline. It consists of experiments performed to build an approach successful and their results. Experiments are conducted to fulfill individual objectives. Results of these experiments are then observed. If there is any problem as an outcome of these experiments, then these problems are dealt until it reaches the proposed aim. This section starts with the requirements of the experiments and then leads to implementation.

3.1 Experiment

A forward pass is required to perform blending/transparency in the deferred rendering pipeline. Forward rendering pass has to be integrated into the deferred pipeline. There are some steps involved to make this work. Simple additions of forward rendering in the deferred rendering pipeline will not suffice. Certain basic implementations need to be implemented before getting to the addition of forward rendering pass. Forward rendering pass is likely to cause problems of getting depth information; frame buffer read and write. After the implementation of generic deferred rendering pipeline and adding forward pass, the interaction between deferred renderer and forward pass needs to have experimented. If any problem outcomes that should be solved to get successful results.

3.2 Requirement

Exercises that are going to be carried out to make this approach successful require some prerequisite steps. After that comes the experiments those will help in solving the problem and lay a systematic implementation of the pipeline. These requirements are listed below with compulsory experiments. Following the list of requirements is the actual implementation details.

- Geometry Pass - Implementation of geometry pass which renders the scene once and then retrieves all the necessary geometry information which is further stored in G-Buffer.
- G-buffer- G- buffer is the major part of geometry pass. It stores the geometry information as textures which is later used by postponed lighting pass.
- The Deferred Lighting Pass - G-buffer provides the significant amount of data fragments to use for lighting calculations. Lighting calculations are done in this pass by iterating the final lighting over every pixel in the geometry scene. All the data provided by G-buffer is then used as input to the lighting algorithms.
- Combining forward rendering Deferred Rendering - Simple way of implementing this is just rendering blending light sources over the scene, but it is not expected to work because forward renderer will not be able to communicate with objects on deferred renderer.
- Communication between deferred renderer and forward pass - Now deferred rendered needs to provide information of all the objects to forward renderer. This step is crucial as all the data can not be passed on to forward renderer.
- Transferring what forward renderer needs - There is a need for the method that will enable the transfer of accurate information only from deferred renderer to forward pass. This experiment is expected to get the proposed outcomes.
- Blending - Test whether the blending works as expected in the formation of approach.
- Light Volumes - Implementation of light volumes instead of general lighting to increase the speed and lessen the lighting computations.

3.2.1 Framework

OpenGL 3.3 is the framework used in this project to implement deferred shading. It was introduced in 1992 (Segal and Akeley 2016). It is considered as an API that allows us to create and manipulate graphics and images (Zhao 2006). It provides with a large set of functions. OpenGL specification exactly specifies what the result of each function would be. It provides freedom to the developer to come with a solution using the specifications of OpenGL.

Most of the previous work on deferred rendering has been done using OpenGL. It has also become the most used gaming interface in the industry for 3D and 2D graphics (ibid.). It seemed the best option for this particular project also because of it allows agility which is required to change shader processing files (Segal and Akeley 2016). The approach in this project requires complete manipulation of basic pipeline including the extension files such as shader. Another main function used in deferred shading is MRT capability of OpenGL. MRTs are used in deferred shading to separate the attributes into textures and write them all collective at once. It is essential for the efficient G buffer construction. There are no limitations to what we can store in the texture in OpenGL.

3.3 Custom Deferred Implementation

3.3.1 Geometry Pass

Now the experiments and requirements of implementations are clear. The OpenGL framework is set up with all the essential libraries such as GL, GLFW, GLM, and ASSIMP. The initial step of implementation would be setting up the geometry pass. Geometry pass has to extract all the information from the objects in the scene and submit that in the g-buffer. This information will be stored in G-Buffer as textures. The four texture used in this project are color vectors, position vectors, normals and specular. These textures can be seen any time in the scene for debugging purposes by changing the mode of the view. As shown in the figure 3.1 below is from one frame.

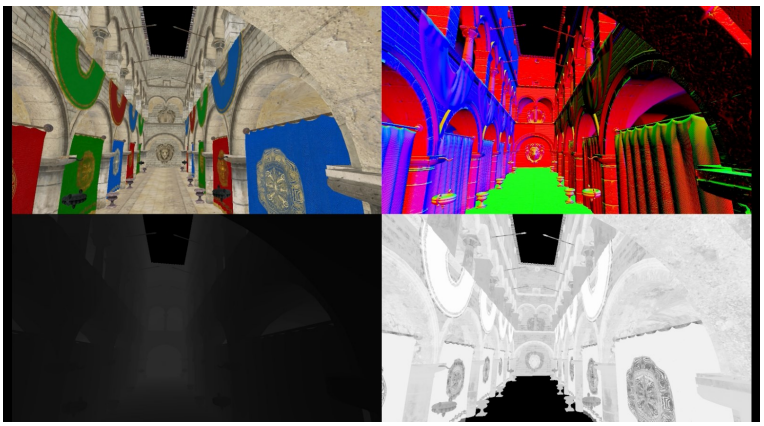

FIGURE 3.1: Visual of textures stored in the G-buffer (Oosten 2015)

As briefly mentioned before, later on, in the pipeline, the screen-filled quad is rendered, and the lighting of the scene is calculated using the G-Buffer data and these textures. Where as in forward rendering we usually take all the objects entire way from vertex to fragment shader. In deferred shading, we separate the expensive fragments to be processed in the later stage. The lighting in lighting stage is similar to forward rendering technique, but instead of taking input from vertex shader we use data from g-buffer. It drastically changes the computations and also the improvement on visuals of rendering can be noticed.

This technique gets rid of most of the unnecessary computations and un-used rendering calls. Depth test is performed at this stage which ensures that the fragments that are stored in the G-Buffer and are the ones to reach the screen as screen pixels. Depth test appoints this fragment's information to the top most fragment. This lead to the main advantage for deferred shading that it allows rendering a large number of lights and objects, which is something not possible in forward rendering.

However, there are certain disadvantages of G-Buffer such as it consumes relatively large memory as it needs to store huge amount of scene information in the color buffers. As mentioned earlier the position vectors are part of G-Buffer. The reason position vectors require more memory be that they have to be high precision. When it comes to the main disadvantage of not supporting blending, the technical reason behind not supporting blending is explained previous sections. So regardless of the fact forward rendering does not allow a large number of lights and objects, but it is useful to perform blending. So, this is the reason we bring back the forward rendering for blending.

3.3.2 G-Buffer

G-buffer development has to be custom as approach aims to integrate forward pass later on. All the textures information collectively are called G-buffer. The list is required of what will be the input in G-buffer, keeping in mind that aim is to render blending with the forward pass. List of data required for lighting calculations for each fragment is as follow

- Position Vector: *lightDir* and *viewDir* functions require 3D position vector to calculate the position variable of a fragment.
- Albedo: An RGB diffused color vector is also required for the fragment.
- Normals: To figure the descent of fragments surface in the scene it requires 3D normals.
- Specular: Float for the specular intensity of the fragment.
- Position for all the light sources and color vectors.
- Position vector for the camera.

Considering this list, availability of these variables make the Blinn-Phong lighting feasible. Blinn-Phong Lighting is the lighting model used in the project. It is more efficient and realistic than Phong lighting. From above list of variables, it can be stated that Position of light sources and color vectors including the position vector for the camera or the viewer can be calculated using uniform variables. Rest of the variables will depend on individual fragments as the different type of fragments will have different intensity and variables. Now the aim is to pass all these or similar to the deferred renderer's lighting pass. One thing to keep in the notice is that deferred renderer is rendering fragments of the 2D quad.

Knowing that OpenGL allows storing as many as textures required, all per fragment data will be retained in these screen filled textures. As proposed, these textures (G-buffer) will store all the variables needed to render blending lights with the forward renderer. This will be the same size as the 2D quad of lighting pass. So pipeline gets the same data as forward renderer but in lighting pass instead of vertex shader to the forward shader process. Deferred rendering has one on one mapping. Below is the entire algorithm in the form of pseudo code for the Geometry pass and lighting pass.

3.3.3 Initial Algorithm

Data: Algorithm
Result: Geometry and Lighting Pass
initialization;
while do
    //1. geometry pass: render all geometric/color data to g-buffer ;
    //join the given frame buffer object to given frame buffer target;
    /celar the color buffer and the depth buffer;
    //use the g shaderbuffer;
    for *Object objec : Objects* **do**
        Configure Shader Transforms And Uniforms();
        objec.Draw();
    end
    // 2. lighting pass: use g-buffer to calculate the scene's lighting ;
    join the given frame buffer object to given frame buffer target;
    clear the color buffer();
    apply the use of lighting shader();
    bind all the textures together();
    set uniform lighting();
    Render the light quads();
end

Algorithm 1: Algorithm of implementation of general deferred rendering

Position, specular, color and normal vector values for each fragment is needed to store in the G-Buffer. All the objects need to be rendered in the Geometry pass and store the values in G-Buffer. The scene is rendered once in the geometry pass. MTRs can be used again to render multiple color buffers in the single pass. G-Buffer is initialized in the geometry pass as frame object. Multiple color buffer and a single depth rendering instance are attached to this frame object. High precision textures are by choice better to calculate position and color textures using 16 or 32-bit float per component. 8-Bit default texture precision per component would be suitable for albedo and specular textures.

In the implementation, first, initiate the G-Buffer then do the mapping calculations for all the four color buffers. Tell OpenGL which color attachment is required for rendering. Then render buffer object as depth buffer is also added and check for completeness. Multiple render targets need to tell OpenGL specific color buffer that is needed to be rendered with glDrawBuffers. The pipeline optimization in this projects has an objective of making the pipeline

highly tolerable to future changes if required. Leading towards this aim, RGB texture store the position and normal data because there are two components each but a single RGBA texture is used to store the specular and color intensity data. This approach saves implementing another color buffer texture. More complex the pipeline gets there is need to combine information in the fewer number of textures to organize the increasing amount of data. Code listing is mentioned in the Appendix A.

Next, there is need to use and render something like fragment shader into the G-Buffer. Again due to the use of MRT, OpenGL is informed about specific color buffer of currently active framebuffer being rendered by the layout specifier. The single float value is stored in the Alpha component of one of the color buffer textures other than specular texture. Specular intensity is not stored in the single color buffer texture. All the coordinates variables are stored in the world space because it is vital to keep them in the same coordinate space for the lighting calculations.

Once the G-buffer frame buffer content is ready, one can visually test the textures according to the origin and RGB color values. To be sure that the calculations are working as expected, the red color should appear more for the normals pointing towards the Red color. It is also same for the position vectors from the origin in the right direction. More details of this test are in Section 4.1.1. Next step after visually testing the rendering of geometry is lighting pass. As it can be seen from the figure 3.2 that G-buffer looks as expected.

FIGURE 3.2: Result of implementation of textures stored in the G-buffer.

3.3.4 Lighting Pass

Now all the data from G-Buffer is available to be used in the lighting pass. It can be easily used to calculate the final lighting colors of the geometry in the scene. It is done by going over each texture in the G-Buffer pixel by pixel. The data obtained from these textures can be finally used efficiently in the proposed lighting algorithm. As afore mentioned that what ever pixel is employed in G-buffer ends up on the screen, so all the final transformation data of objects in the scene is already available. As known that lighting is the most expensive computation, now its only has to be performed once per pixel. This method also explains how deferred shading becomes more efficient when the scene gets more complex with a large number of object and lights. Such as in forwarding rendering, renderer calls the expensive fragment shaders multiple times which makes the performance less efficient in regards to speed.

The 2D screen filled quad is rendered to execute the lighting pass. On each pixel an extensive just like the post processing effect. Before the rendering, the scene, light relevant uniform variables are sent to the shader. All textures present in the G-Buffer also needs to bound before rendering. Below is the short code listing 3.1 of how it is done.

Implementation of fragment shader can be seen in Appendix B. The fragment

LISTING 3.1: Binding textures and applying fragment shader to Each Pixel

```
gl_Clear(GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT);
//activate first texture0
gl_BindTexture(GL TEXTURE 2D, gPosition_);
//activate second texture1
gl_BindTexture(GL TEXTURE 2D, gNormal_);
//activate third texture2
gl_BindTexture(GL TEXTURE 2D, gAlbedoSpec_);
// also send light relevant uniforms
shaderLightingPass.use();
\\send all the lighting uniforms(shaderLightingPass);
shader Lighting Pass.set_Vec3("view_Pos", camera.Position);
\\RENDER THE LIGHT QUADS();
```

shader formations in Appendix B is very similar to the one in state of the art forward rendering but instead of using vertex shader, data from G-Buffer in directly sampled as an input to the fragment shader.

Three uniforms textures are accepted by shader of lighting pass. This representation of G-Buffer tenures all the data of Geometry Pass. Having lighting pass shader and geometry pass fragment shader, the coordinates of both should match exactly when sampled together. These will output the same coordinate values similar to what they will produce if they were rendered directly. Referring again to the fragment shader variables relevant to lighting pass are extracted from the shader. This extraction is done by using simple texture lookup. Albedo and specular intensity are extracted from a single texture. It is named *gAlbedoSpec* texture in the implementation.

Implementation so far has necessary per fragment variables and also have relevant uniform variables. These variables are enough to calculate Bling-Phong Lighting. So the implementation code will be very similar to a general Bling-Phong lighting implementation. Deferred rendering does differ in the way of receiving input variables for illumination. The image below in figure 3.3 is the resulting outcome of deferred shading implementation so far.

FIGURE 3.3: Result of implementation deferred rendering so far without blending

3.3.5 Limitation of this implementation

Now it comes to the main objective of this project. Limitations of this approach so far are same as a generic deferred pipeline. Deferred shading so far will not allow blending. All the elements in G-Buffer are from a single fragment. To render blending it requires multiple fragments. As explained in detail previously in the research question section 2.1 is the reason behind this limitation. As a solution, we will bring forward rendering back due to its capability of rendering blending as proposed.

3.4 Forward Pass integration into Deferred Pipeline

The original algorithm will be amended to eradicate the problem of blending. Rendering pipeline has to integrate forward rendering pass. This forward rendering pass will specifically render the blending or any other special shader that is not supported by deferred rendering. This method will take the deferred rendering technique a step up. The forward pass will include a number small light sources. These light sources will emit color on to the objects in the scene. These light sources require special shader which is not supported by deferred shading. So they will be rendered used forward pass.

3.4.1 Algorithm Improvisation 1

Below is the brief detail of algorithm. This algorithm is derived from previous algorithm 1 in section 3.3.3. It is altered and improved to solve the problem of blending by adding forward pass in the pipeline to render colored light sources. Notice that some of the steps are minimized. Refer to algorithm 2 for these steps.

Data: 1st Improvisation in Algorithm
Result: Geometry, Lighting and Forwards Pass
initialization;
while do
    //1. geometry pass: render all geometric/color data to g-buffer
    ..
    more steps here
    ..
    for *Object obj : Objects* **do**
        ConfigureShaderTransformsAndUniforms();
        obj.Draw();
    end

    // 2. lighting pass: use g-buffer to calculate the scene's lighting
    ..
    more steps here
    ..
    RenderQuad();

    // 3. Forward Pass: To render colored light sources with special shaders
    shader LightBox.use()
    shader LightBox.setMat4("projection", projection)
    shader LightBox.setMat4("view", view)
    for *unsigned int i = 0; i < lightPosition.andsize(); i++* **do**
        //All the Modeling data; shader LightBox. setMat4("model", model)
        shader LightBox. setVec3("light Color", light Colors[i])
        render the light cubes();
    end
end

Algorithm 2: Improvised Algorithm with addition of Forward Pass

Rendering the colored cubes as colored light sources emitting the light on

to the deferred rendered objects is the next step. Simple implementation such as above algorithm is the first experiment. Rest of the implementation depends on the outcome of this experiment. So lets render these colored cubes over the 2D quad of deferred renderer. This will be implemented after the lighting pass in the main rendering loop as stated in the improvised algorithm. The positions of these cubes are calculated and number of lights (cubes) are extracted from the fragment input. It is equal to the number of lights rendered. After running this experiment and rendering the scene with these colored light cubes provides the results as shown in figure 3.4 below.

FIGURE 3.4: First experiment to integrate forward pass to deferred pipeline

From the visual look at the results, it is easily noticeable that there is no blending happening in the scene. Even though the colored light cubes are rendered on top of the geometry, the colored light effect is not present. The reason behind that is missing Depth information. Forward pass does not have any information about the depth data of the geometry in the deferred renderer. There is a lack of interaction between deferred passes and the forward pass.

3.4.2 Interaction between existing Deferred Renderer and Forward pass

The second experiment is to have a default frame buffer for both of the renderer in common is the. Depth information needs to be transferred from buffer in geometry pass to depth buffer of default frame buffer. Colored light sources can then be rendered again. If only depth data of geometry is transferred then the colored light will only affect the geometry already rendered.

After implementing this experiment, the results move step forward but is still not what is expected according to the proposed outcome. All the depth data is transferred to default frame buffer expecting better results by rendering light cubes again. Now the whole scene appears to be rendered with the forward rendering including the geometry already rendered with the deferred renderer. This outcome was not expected from the technical approach. It certainly needs a fix otherwise implementation looses its essential positive characteristics of deferred rendering. Deferred rendering will not be serving its purpose of rendering a large number of objects and lights.

The function that comes in use at this point is *glBlitFramebuffer*. This function allows copying the data from one frame buffer to another frame buffer. It is also used to resolve multi-sampled frame buffers. Copying the specific user defined regions of the frame buffer to the specific area of another frame buffer becomes possible with *glBlitFramebuffer* function.

All the depth data values of geometry are stored in deferred renderer's shading pass G-Buffer. Use of the function *glBlitFramebuffer* requires specifying one frame buffer as read and another frame buffer as write frame buffer to avoid the negative outcome of the second experiment. Following this method, entire read frame buffer's depth buffer is copied to default frame buffer's depth frame buffer. The Same approach can be used to copy the data from color and stencil buffers using *glBlitFramebuffer* function.

3.4.3 Algorithm Improvisation 2

Now algorithm changes slightly due to the outcomes of experiments conducted. To implement forward pass the difference or changes from the last algorithm are shown below in algorithm, and other steps are minimized. For

minimized step refer to algorithm 2 in section 3.4.1. Next is algorithm 3 with

specific read and write frame buffers.

Data: Algorithm Improvisation 2
Result: Specific Read and write Frame Buffers
;
while do
    //1. geometry pass: render all geometric/color data to g-buffer
    ..
    more steps here
    ..
    for *Object obj : Objects* **do**
        ConfigureShaderTransformsAndUniforms();
        obj.Draw();
    end

    // 2. lighting pass: use g-buffer to calculate the scene's lighting
    ..
    more steps here
    ..
    RenderQuad();

    //3. Copying depth data from read frame buffer to default frame buffer
    bind the framebuffer()(GL READ FRAMEBUFFER, gBuffer);
    g*bind the framebuffer()*(GL DRAW FRAMEBUFFER, 0) *// write to default framebuffer* ;
    use gl blit function here(0, 0, SCREEN WIDTH, SCREEN HEIGHT, 0, 0, SCREEN WIDTH, SCREEN HEIGHT, GL DEPTH BUFFER BIT, GL NEAREST);
    glBindFramebuffer(GL FRAMEBUFFER, 0);

    // 4. Forward Pass: To render colored light sources with special shaders
    shaderLightBox.use()
    for *unsigned int i = 0; i < lightPositions.size(); i++* **do**
        / / now render light cubes as before
        ..
        more steps here
        ..
        RenderCube();
    end
end

Algorithm 3: Algorithm Improvisation 2 showing read and write frame buffer

Rendering this implementation produces more realistic results. Blending works and outputs desired results. The drawback of the last experiment where all the geometry looked like rendered with the forward renderer has been eliminated. At this point, it outputs the proposed result. The outcome from the demo scene can be seen in the figure 3.5.

FIGURE 3.5: Deferred rendering and blending with right depths.

Till here this technique has integrated forward pass into the deferred rendering pipeline. It takes out the limitation of blending and rendering objects with special shaders.

3.5 Light Volumes

Furthermore is the implementation of light volumes. To conclude the approach with full efficiency, light volume calculation is crucial. Light volume makes the deferred pipeline more efficient in rendering a large number of lights by reducing the area for the fragment calculations. This implementation will allow rendering huge number of light without putting any burden on the performance. If this is left out from the pipeline then still each fragment of light will have to be calculated for each object in the scene including the object not even affected by that particular light. As explained in the background research, there is a need to get rid of unnecessary computation. These unnecessary computations for the lighting calculation are the calculation performed

for the fragments in the scene till where the light does not even reach. So, lighting calculations should have a certain distance for every light up to the point that light matters. The objective here is to find out this distance which in the case of lights will be radius or volume of any light source. Some attenuation is used by most of the lights in the rendering process. This attenuation can be used to know the maximum distance or the radius that particular light reaches. By knowing this limit, computation performed by lighting calculation can also be limited. Then only the fragments those come under one or more of this radius of light volumes will be calculated.

The key is the calculation of the radius of spread light of any light source. As afore mentioned attenuation equation needs to be solved for this case. The level of brightness is from 0.0 to 1.0 where 0 being the darkest and 1 being the brightest value. O is the lowest value, but something like 0.3 will also be pretty bleak. The appropriate function of attenuation will be used to calculate this. The equation for this function is as mentioned in section 2.2.4 is

$$F_{light} = \frac{1}{K_c + K_l * d + K_q * d^2}$$

Therefore, the value of F_{light} needs to be calculated for the darkness. The lowest value for brightness would usually be zero, but in this equation, F_{light} it can not be zero. If it is tried to be solved for F_{light} = 0.0 then there will be no solution. Any value closer to 0.0 that is still precipitable as dark enough can be used to solve the equation. A default frame buffer of 8 bit can display 256 number of entities per each component. So the number adequate for this project is 5/256.

Note that this number can be changed according to the requirements of the project. So this is feasible for the customization for other implementations. Attenuation function value will take off the sudden loss of light at the edges of the light volume. The value is essentially dark in its visible field. For this particular demo, if the value of darkness is calculated for an even darker area it will become less efficient as the light volumes will get bigger. So depending on the scene and its visual implementation this number will vary and so will the performance. However, the main aim is to make the lights more realistic and get the perfect blend going from high-density end to low density of light source.

Deriving from previous attenuation equation

$$F_{light} = \frac{1}{K_c + K_l * d + K_q * d^2}$$

Therefore, after the input of F_{light} value to $\frac{5}{256}$, equation becomes,

$$\frac{5}{256} = \frac{I_{max}}{Attenuation}$$

Usage of light source's brightest component value outputs more realistic and ideal results for the light volume calculation. That is why the I_{max} is used. I_{max} is the variable for the lights highest brightness component value. Following this is the rest of the derivation of this equation.

$$\frac{5}{256} * Attenuation = I_{max}$$

$$5 * Attenuation = I_{max} * 256$$

$$Attenuation = I_{max} * \frac{256}{5}$$

$$K_c + K_l * d + K_q * d^2 = I_{max} * \frac{256}{5}$$

$$K_q * d^2 + K_l * d + K_c - I_{max} * \frac{256}{5} = 0$$

Now the form of this equation is equals to $ax^2 + bx + c = 0$. To further explain $x = d$ so, $ax^2 = K_q * d^2$, $bx = K_l * d$, $c = K_c - I_{max} * \frac{256}{5}$. Hence, for this type of equation the solution is using quadratic equation. The solutions turn out to be the following equations for the value of x.

$$x = \frac{-K_l + \sqrt{K_l^2 - 4 * K_q * (K_c - I_{max} * \frac{256}{5})}}{2 * K_q}$$

Therefore from this outcome, calculation of x can be done. x is the radius of the light source's volume. This radius requires a liner, quadratic and a constant parameter to solve the equation. Input for these variables is used as such $floatconstant = 1.0$, $floatlinear = 0.7$, $floatquadratic = 1.8$. This general equation then put into code will look something like this

LISTING 3.2: Light volume calculation code Listing

```
1
2 float maximumlight = std::fmaxf(std::fmaxf(lightingColor.r, lightingColor.g), lightingColor.b);
```

```
3 float radius =
4   (−linear + std:: sqrtf (linearterm ∗ linearterm − 4 ∗ quadraticterm ∗ (constantterm − (256.0 / 5.0) ∗ maxi
5   / (2 ∗ quadraticterm);
```

According to this, the output value of radius somewhere is between 1.0 to 5.0. This result is established depending on maximum light intensity. Now having the light volumes calculated, light pass fragment shader needs to be updated. The radius of each light source present in the scene is computed. It is used in doing the lighting calculations for each fragment of objects that come under these light volumes. The update light pass fragment shader is placed in appendix C.

Next in figure 3.6 is the outcome of rendering using the light volumes. It will visually look same but the number of computations is lessened. It does have some limitations that need work around. More detail about these confinements is mentioned in coming sections of this report.

FIGURE 3.6: Deferred rendering and blending using Light Volumes.

Chapter 4

Evaluation and Results

It has been proved through experiments and implementation that the proposed approach produces the desired results. The technique allows the deferred rendering pipeline to perform blending. Implementation had some constructive changes in the algorithm to reach the stage where blending is possible. The decisions were made on the results of experiments and the reason behind the results. Following Blending implementation is a very smart optimization of Light Volumes to the pipeline. All these aspects are implemented and demonstrated working but how efficiently they work is still needs to be evaluated. The decision made to optimize the pipeline and changes need to be reviewed. This section will focus on the evaluation of these decisions in implementation and record the results. For testing the approach, hardware being used is Intel(R) Core(TM) i5-3210M CPU @2.50GHz and usable 5.86GB RAM. The operating system is 64-Bit Windows 10 with the x64-based processor. The specification of hardware is a mediocre level. The hardware choice is made to cover the vast spectrum of users as per the aim of the approach.

4.1 Evaluation

4.1.1 Geometry Pass textures

The first most of implementing the deferred pipeline is geometry pass. G-buffer frame buffer should be working properly to proceed to the next step. If it's not right, the results won't be appreciable. The result of geometry pass can be tested visually by comparing it with *CIE RGB color space diagram in 2D and 3D* in figure 4.1. (Wright 1929) and (Guild 1932) conducted experiments independently on human sight. These experiments are the principle of the

specification of *CIE XYZ Color Space*. This color space links to the psychological color vision on human perception.

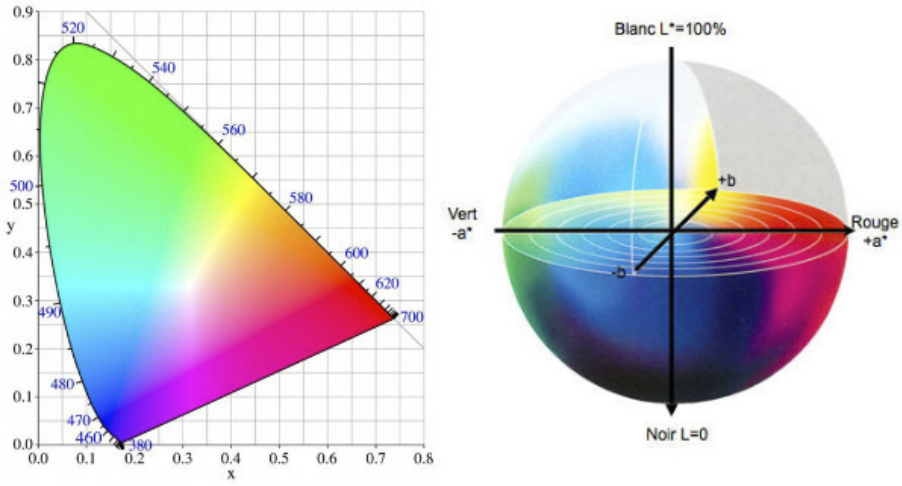

FIGURE 4.1: CIE RGB color space diagram in 2D and 3D (Frich 2017)

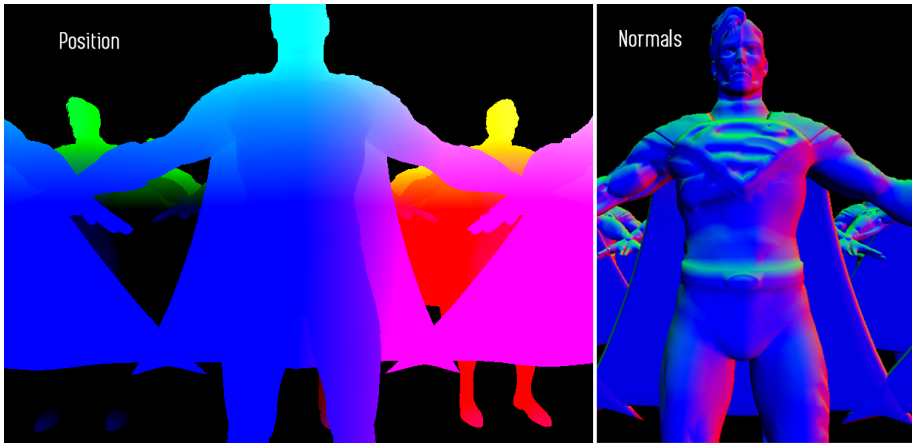

FIGURE 4.2: G-Buffer frame buffer textures colors and normal.

G-buffer frame buffer's texture of position and normals texture data should follow the directions of colors in the color space. Figure 4.1 and figure 4.2 can be visually compared. Having origin in the center $x+$ should have pink in its direction, and $x-$ should have blue in its direction. Similarly, following the pattern in all directions, it should be green in the left back end and orange on the right side. Looking at figure 4.1 and figure 4.2, one can easily state that texture has been rendered correctly. Implementation can proceed to the next step.

4.1.2 Performance Analysis

Implementation output the proposed features as seen in previous sections. Data collected from performance analysis is discussed in forthcoming parts of this section. Test on this implementation was performed to check how efficiently the approach handles the increase in the number of objects and number of lights in the scene. Data is recorded in table 4.1. A total of four different scenarios were tested to measure the performance fluctuations.

File size for each of these variations was approximately 20.65MB after the compression with the ratio of 2.69:1. Draw calls for each of the variation changes accordingly. There are 11 different textures present in each of the rendering variation. These textures comprise to 25.02MB. Avg. texture dimension comes out to be 896 x 802. There are 29 Buffers each test which in total are 3.69MB. All together additively, the grand total of GPU buffer plus texture load it becomes 48.92471MB of data rendered. File size: 20.65MB (55.52MB uncompressed, compression ratio 2.69:1) Persistent Data (approx): 0.05MB, Frame-initial data (approx): 35.60MB

Performance Analysis						
ID	No. Of Objects	No. Of Lights	Min Frame Time (ms)	Max Frame Time (ms)	Avg. Frame Rate	Draw Calls
1	2	4	06.25	10.73	106 Fps	15
2	9	32	17.95	23.51	50 Fps	78
3	18	64	20.52	27.08	45 Fps	155
4	36	128	23.95	29.51	40 Fps	309

TABLE 4.1: Performance Analysis table for different scale of objects and lights.

Leading towards the results collected after the rendering test. There are four different scenarios listed in table 4.1. From ID 2 to 4, the number of lights and objects were almost doubled every time. ID 1 and 2 have a big difference between them in comparison to other increments. The first case is with 2 objects and 4 colored lights that perform blending. Minimum and maximum frame time in ms are 06.25 and 10.73 respectively which is very reasonable and efficient. Similarly, the frame rate, in this case, is an average of 106 Fps, which again is high and suitable for rendering. The draw calls are only 15 in this case.

Secondly, next variation is a slightly a big jump from the 1st case. Now the number of objects is 9 and lights are 32. This rendering utilizes 78 draw calls

to render this scene. Now the number of minimum and maximum frame time is 17.95ms and 23.52ms respectively. This hike in frame time is quite huge from the 1st case. It output 50 frames per sec which are approximately half in comparison to the 1st case. Still, it can not be considered bad because 50fps still provides satisfactory performance.

In third variation lights and objects are scaled to double of the second case. Now there are 18 objects and 64 number of lights. This predictively uses 155 draw calls twice of the last variation. Frame time in this scenario increases slightly by approximately 3ms. This result is not bad considering the hike in the number of objects and lights. 5 frames per sec is the drop recorded in frames per sec. Again not a big drop in comparison to the scaling of objects and lights.

Fourth and last test case had 36 objects and 128 lights. These numbers are high to check the change in performance. In hand to hand with the rise in objects and lights, draw calls are again approximately doubled to 309. Frame time gained even lesser this time by only 2ms. Frames per sec dropped by average five frames.

Overview of these performance test analysis provides insight on how efficiency changes in this particular optimization to the deferred rendering pipeline. Performance degrades massively when the number of objects and lights are increased from little number to an average number. But as the number of objects keeps getting higher the performance drop gradually decreases. This resulting fact means the objective of rendering a large number of objects and lights begin to justify it self with the increase in these figures. So if there are a large number of objects and lights, this approach will still provide satisfactory performance.

GPU and CPU utilization variation test

Next test performed was to measure the CPU and GPU utilization by the rendering application. This test was only observed in the two extreme cases with case ID 1 and 4. As afore mentioned the first instance is with 2 objects and 4 lights. The second case consists of 36 objects and 128 lights. Data for this test is stored in the table 4.2. This test is performed illustrates that the change in CPU and GPU utilization percentage when the number increases from tiny scale to large scale.

The first extreme is the smaller number of 2 objects and 4 lights where CPU utilization is 3 percent from the available 100 percent. GPU usage for this rendering applications is 98.3 percent. When the number of objects increases to a very high number as in case ID 4 with 36 objects and 128 lights, the result of utilization does not change much. The percentage of CPU increased to 8 percent, and GPU is increased to 99 percent. This difference is almost negligible considering the difference between the number of lights and objects. This state that computation behind the scene rendering does not change much with these variations.

CPU and GPU Utilization				
ID	No. Of Objects	No. Of Lights	CPU (%)	GPU (%)
1	2	4	3	98.3
4	36	128	8	99

TABLE 4.2: Table of CPU and GPU Utilization For Variation of Object and Light Number

4.1.3 Light Volumes Vs No Light Volumes

Light Volume calculations justify themselves. The Visual result after implementing light volumes is exactly same as before. To review the results of Light Volume implementation it needs to be evaluated using some form of metric. There needs to be a comparison between rendering performance without Light Volumes and after the implementation of Light Volume.

For this evaluation, by comparison, two stages of implementation were rendered independently with the same number of objects and lights. The variable used to compare the performance are CPU Utilization for the application, Frames per second, Frame Time and memory usage. The CPU usage time line graph that compares the total activity of the rendered console application is to total available CPU capacity percentage. CPU utilization is calculated using all the processors available. Frame per sec is how many frames does the renderer can output in a sec. After running each rendering several time for approximately 1 minute, the average frame per sec calculated is used in the evaluation. Frames per sec are calculated by rendering nearly over 8,000 frames for each application. Then is the memory usage by the process in percentage. Faster rendering will output more frames. The frame time is

inversely proportional to the speed of rendering. Frame time needs to be a lower number for faster rendering.

Without Light Volumes

Application rendered without Light Volume calculation is fully working. It did not out put slow results. It visually has the same look as rendering with Light Volumes. As shown in Table 4.3, tested using FPS and FT as a metric gives out minimum frame time to be 23.52ms and maximum frame time is 27.08ms. This calculation of frame time is done over approx. 8,000 frames. The average frame rate depicted by the application is 40 frame per sec. Again this number is calculated by running the application for several numbers of times.

Adding further on to performance analysis for implementation before Light volumes calculation is the data from GPU and CPU utilization. The GPU usage percentage is staying mostly consistent through out the application's running period. GPU utilization once the application is loaded and running stays at average 98.5 percent. CPU usage keeps fluctuating. When the application is being loaded goes up to approx. 25 percent. Once the console is running the utilization stays at an average of 5 percent. As it can be seen from the graph in Figure 4.3.

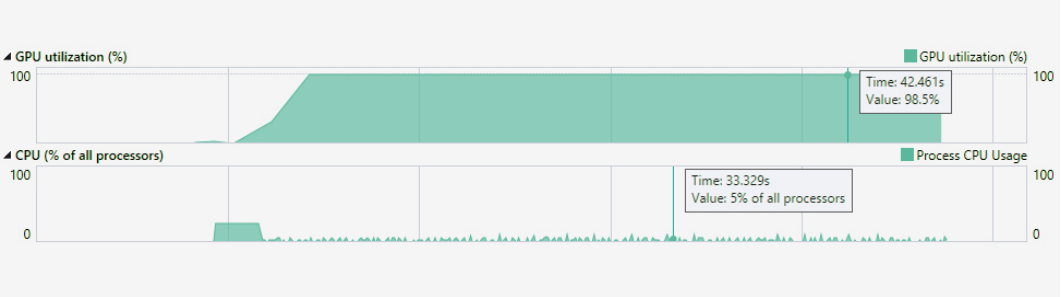

FIGURE 4.3: Graph of GPU and CPU utilization without Light Volumes.

With Light Volumes

Light Volume Implementation aimed at reducing computational cost by limiting the calculations for each light to the volume that matters for that light. After implementing the light volumes, the performance analysis is recorded as shown in Table 4.3. The minimum frame time is in milliseconds is 17.95,

and the maximum is 20.51ms over approximately 8,000 frames. The average frame rate of the application is recorded to 53 frames per sec.

Light volume implementation rendered with the same number of objects and lights utilizes GPU and CPU as shown in Figure 4.4. The graph on the top shows GPU utilization to be consistent thought out just like before. The average percentage is 98.3 percent. The graph below shows the CPU utilization percentage. While loading the console and start rendering, CPU usage goes up to 25 percent. While rendering the application with the light volume calculation, CPU utilization is at an average of 4 percent.

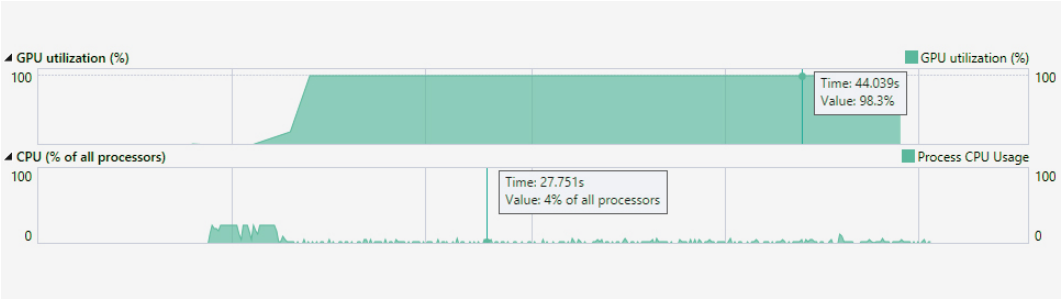

FIGURE 4.4: Graph of GPU and CPU utilization after Light Volumes.

Performance Comparison With and Without Light Volumes			
Rendering Application	Min Frame Time (ms)	Max Frame Time (ms)	Avg. Frame Rate
Without Light Volumes	23.52	27.08	40 Fps
With Light Volumes	17.95	20.51	53 Fps

TABLE 4.3: Table of performance comparison between With and Without Light Volumes

Following results are from comparing the data from these two implementations. The difference can be stated quickly in case of Frame per sec and Frame time analysis. Frame time reduces approximately 7ms from both minimum and maximum values. Average frames per sec improve by 13Fps. So it can be concluded that performance does improve from generic deferred shading by using light volumes, but there is no big difference.

Along with that, performance profiling of CPU and GPU utilization does not show much difference. Approximately both scenarios are close to each other. Variation in GPU utilization is only 0.5 percent. Similarly, CPU usage is only 1 percent less in Light volumes implementation than before.

4.2 Limitations

This section shred light on few limitation of this approach and what is the reason behind it. Most of the results of this approach and implementation are satisfactory, but still, there are some thing that can be improved. Performance analysis results are justifying the objective of the proposition, but they are still affected by light volumes and depend on it. The results of the light volume are not up to the expectations. The theory and implementation support the argument and lead to the high level of optimization it can. The problem behind this is that GPU and GLSL are not efficient enough at optimizing loops and branches in the commands. Shader execution of graphic hardware is notably parallel. To have the same shader code to be effective, most of the architectures require a huge set of thread to run this shader code. This way it mostly ends up with shader running all branches of a *if* statement. It does so to make sure that shader runs are identical. This limitation makes our implementation of light volumes very much inefficient regardless of having the right approach and calculations. There is a way around this that can beneficial to over come this limitation. The solution to this problem is suggested in future work section 5.2.

Chapter 5

Conclusion

5.1 Conclusion

In the nut shell, this approach aimed at optimizing deferred rendering pipeline so that new version can get rid of limitation of generic pipeline and make it faster. Firstly implementing the customized form of generic deferred rendering pipeline is successful done. The most common limitation of the deferred rendering is blending. To deal with blending this approach integrates the forward pass in the deferred rendering pipeline successfully after a couple of experiments and changes. Getting that done solves the problem of blending or transparency. Further more to complete the optimization and make a fully working technique, Light volumes concept is used to change the way the light calculation is done for objects in the scene.

However, the light volume calculation does not work as expected due to the external reason of compatibility as mentioned in limitations section. Light volume does not put any heavy load on the processing of the application, it, in fact, is always faster than the generic deferred pipeline. It can be worked around using the methodology stated in the future work section coming next in the report.

Overall, this approach fulfills its proposed aims and objective and successfully implemented them. This method can be used on the vast scale and variations of projects. It can be a start off point for any project which intends to use the deferred rendering pipeline as its core for rendering. This approach will support customization at best. It can be altered and scaled according to the requirements of the graphics and speed. For example, if the user wants more efficient light volume calculations then the methodology mention in future work can be implemented with enabled Face culling. It will be highly robust

and faster that way. Similarly, the developer can extend this approach or use it as it is, depending on the type of project. In the end, with all the evidence from the evaluation and testing, this project can be concluded as success with an opportunity to make it more advance in the future.

5.2 Future Work

Further more in the future, light volume implementation can be optimized to be fully effective. After the implementation of the radius and light attenuation, it is supposed to give out the better result with significant changes. After the performance analysis and testing of this approach, the reason behind not getting these expected results was found out. Incompetency of GPU architecture and GLSL at optimizing loop and branches is the reason.

The more befitting technique to implement light volumes would be slightly different. The actual sphere should be rendered with the light source at the center of this sphere, using the same calculation of attenuation. It should be scaled equally to the light volume radius such as way that it encloses the entire volume of light. Familiarizing, we can use the same shader of deferred fragment shader and render these spheres with it.

Having the spheres calling the same deferred fragment shader will match the pieces that are affected by the light source. Hence, what ever pixel comes in this sphere should be rendered ignoring all other pixels. If this is implemented for all the number of lights, it will reduce the computations drastically. Instead of doing $numberofobject * numberoflight$ the calculations will become $numberofobjects + numberoflights$. This improvisation will make the optimization complete for deferred rendering pipeline making it adequately efficient.

Appendix A

Code Listing for Rendering all colorbuffers in to G-Buffer

```
1
2  layout_ (location0) out vec3 gPosition;
3  layout_ (location1) out vec3 gNormal;
4  layout_ (location2) out vec4 gAlbedoSpec;
5
6  in vec2 Tex_Coords;
7  in vec3 Frag_Pos;
8  in vec3 Normal;
9
10 uniform sampler2D texture diffuse_1;
11 uniform sampler2D texture specular_1;
12
13 void main()
14 {
15     // fragment position vector in the first gbuffer texture
16     gPosition = FragPos;
17     // the per-fragment normals into the gbuffer
18     gNormal = normalize(Normal);
19     // diffuse per-fragment color
20     gAlbedoSpec.rgb = texture(texture_diffuse1, TexCoords).rgb;
21     // specular intensity in gAlbedoSpec's alpha component
22     gAlbedoSpec.a = texture(texture_specular1, TexCoords).r;
23 }
```

Appendix B

Code Listing Of Frame Buffer directly sampled from G-Buffer

```
#version 330 core
out vec_4 Frag_Color;

in vec_2 Tex_Coords;

uniform sampler2D g_Position;
uniform sampler2D g_Normal;
uniform sampler2D g_AlbedoSpec;

struct Light_ {
    vec3 Position_;
    vec3 Color_;
};
const int NR_LIGHTS = 800;
uniform Light lights[NR_LIGHTS];
uniform vec3 viewPos;

void main()
{
    // retrieve data from G−buffer
    vec3 FragPos_ = texture(g_Position, Tex_Coords).rgb;
    vec3 Normal_ = texture(g_Normal, Tex_Coords).rgb;
    vec3 Albedo_ = texture(g_AlbedoSpec, Tex_Coords).rgb;
    float Specular_ = texture(g_AlbedoSpec, Tex_Coords).a;

    // then calculate lighting as usual
    vec_3 lighting_ = Albedo_ * 0.1; // hard−coded ambient component
    vec_3 view_Dir = normalize(view_Pos − Frag_Pos);
```

```
29      for(int i = 0; i < NR_LIGHTS; ++i)
30      {
31          // diffuse
32          vec3 light_Dir = normalize_(lights_[i].Position_ − Frag_Pos);
33          vec3 diffuse = max(dot(Normal_, light_Dir), 0.0) ∗ Albedo_ ∗ lights_ [i].Color_;
34          lighting_ += diffuse_;
35      }
36
37      Frag_Color = vec4(lighting_, 1.0);
38  }
```

Appendix C

Code Listing Updated Lighting pass fragment shader

```
1  struct Light {
2      [...] //more steps here
3      float Radius;
4  };
5
6  void main()
7  {
8      [...] more steps here
9      for(int i = 0; i < Number_of_LIGHTS; ++i)
10     {
11         // calculate the distance between light source and fragment that is affected by that light
12         float distance_ = length(lights[i].Position_ − Frag_Pos);
13         if(distance_ < lights[i].Radius_)
14         {
15             // do the expensive lighting
16             [...] more steps here
17         }
18     }
19 }
```

Appendix D

Code and Document Link

Following is the web address for entire projects code files and documents.
http://gitlab.scss.tcd.ie/sakumar/dissertation.git

Bibliography

Andersson, Johan (2009). *Parallel graphics in frostbite - current future*. SIGGRAPH Course Beyond Programmable Shading.

Balestra, Christophe and Pal-Kristian Engstad (2008). *The Technology of Uncharted: Drake's Fortune*. Game Developer Conference.

Billeter, Markus, Ola Olsson, and Ulf Assarsson (2009). "Efficient Stream Compaction on Wide SIMD Many-core Architectures". In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: ACM, pp. 159–166. ISBN: 978-1-60558-603-8.

Blinn, James F. (1977). "Models of Light Reflection for Computer Synthesized Pictures". In: *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '77. San Jose, California: ACM, pp. 192–198.

– (1998). "Seminal Graphics". In: New York, NY, USA: ACM. Chap. Models of Light Reflection for Computer Synthesized Pictures, pp. 103–109. ISBN: 1-58113-052-X.

Chajdas, Matthäus G., Morgan McGuire, and David Luebke (2011). "Subpixel Reconstruction Antialiasing for Deferred Shading". In: *Symposium on Interactive 3D Graphics and Games*. I3D '11. San Francisco, California: ACM, 15–22 PAGE@7. ISBN: 978-1-4503-0565-5.

Cohen, L. G. (1971). "Measured attenuation and depolarization of light transmitted along glass fibers". In: *The Bell System Technical Journal* 50.1, pp. 23–42. ISSN: 0005-8580.

Crytek (2010). *Cryengine3 sponza model*. URL: http://www.crytek.com/cryengine/cryengine3/downloads.

Deering, Michael et al. (1988). "The triangle processor and normal vector shader". In: *88 Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. New York: SIGGRAPH, pp. 21–30.

Frich, Arnaud (2017). *Color Management Guide*. [Online; accessed July 28, 2017]. URL: http://www.color-management-guide.com/color-spaces.html.

Fuchs, Henry et al. (1989). "Pixel-planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-enhanced Memories". In: *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '89. New York, NY, USA: ACM, pp. 79–88. ISBN: 0-89791-312-4.

Guild, J. (1932). "The Colorimetric Properties of the Spectrum". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 230.681-693, pp. 149–187. ISSN: 0264-3952.

Hargreaves, Shawn (2010). *Deferred Shading*. Game Developers Conference.

Hoberock J, Bell N (2010). *Thrust: A parallel template library*. Nvidia. URL: http://www.meganewtons.com/.%204,%207.

Jaud, Thomas, Anne-CÃ©cile Dragon, et al. (2012). *Relationship between Chlorophyll a Concentration, Light Attenuation and Diving Depth of the Southern Elephant Seal Mirounga leonina*. [Online; accessed July 10, 2017]. URL: https://doi.org/10.1371/journal.pone.0047444.

Jaud, Thomas, Anne-Caocile Dragon, et al. (2012). "Relationship between Chlorophyll a Concentration, Light Attenuation and Diving Depth of the Southern Elephant Seal Mirounga leonina". In: *PLOS ONE* 7, pp. 1–10.

Kircher, Scott and Alan Lawrance (2009). "Inferred Lighting: Fast Dynamic Lighting and Shadows for Opaque and Translucent Objects". In: *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*. Sandbox '09. New Orleans, Louisiana: ACM, pp. 39–45. ISBN: 978-1-60558-514-7.

Lauritzen, Andrew (2010). *Deferred rendering for current and future rendering pipelines*. SIGGRAPH Course: Beyond Programmable Shading.

Magnerfelt, Christian (2012a). "Transparency with Deferred Shading". Viewed 01 June 2017. MA thesis. Royal Institute of Technology.

– (2012b). *Transparency with Deferred Shading*. [Online; accessed June 25, 2017]. URL: http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group1Marten/report/report_magnerfelt.pdf.

Markager, Stiig and Warwick F Vincent (2000). "Spectral light attenuation and the absorption of UV and blue light in natural waters". In: *Limnology and Oceanography* 45.3, pp. 642–650.

McGuire, Morgan and Louis Bavoil (2013). "Weighted Blended Order-Independent Transparency". In: *Journal of Computer Graphics Techniques (JCGT)* 2.2, pp. 122–141. ISSN: 2331-7418.

Olsson, Ola and Ulf Assarsson (2011a). *Tiled Shading*. [Online; accessed June 12, 2017]. URL: http://dx.doi.org/10.1080/2151237X.2011.621761.

– (2011b). "Tiled Shading". In: *Journal of Graphics, GPU, and Game Tools* 15.4, pp. 235–251.

Olsson, Ola, Markus Billeter, and Ulf Assarsson (2012a). "Clustered Deferred and Forward Shading". In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG'12. Paris, France: Eurographics Association, pp. 87–96. ISBN: 978-3-905674-41-5.

– (2012b). "Tiled and clustered forward shading". In: *Memory* 300.400, p. 500.

Oosten, Jeremiah Van (2015). *Forward vs Deferred vs Forward+ Rendering with DirectX 11*. [Online; accessed July 25, 2017]. URL: https://www.3dgep.com/forward-plus/.

Phong, Bui Tuong (1975). "Illumination for Computer Generated Pictures". In: *Commun. ACM* 18.6, pp. 311–317. ISSN: 0001-0782.

Policarpo, Fabio, Francisco Fonseca, and Checkmate Games (2005). *Deferred Shading Tutorial*.

Segal, Mark and Kurt Akeley (2016). *The OpenGL Graphics Interface*. Tech. rep. SILICON GRAPHICS COMPUTER SYSTEMS.

Swoboda, Matt (2009). *Deferred lighting and post processing on playstation 3*. Game Developer Conference.

Thaler, Jonathan and TU Wien (2010). "Deferred Rendering". In: Austria: Vienna University of Technology, pp. 1–2.

Unreal (2011). *Unreal development kit*. URL: http://www.udk.com/.

Wright, W D (1929). "A re-determination of the trichromatic coefficients of the spectral colours". In: *Transactions of the Optical Society* 30.4, p. 141.

Zhang, Yanci and Renato Pajarola (2007). "Deferred blending: Image composition for single-pass point rendering". In: *Computers & Graphics* 31.2, pp. 175–189. ISSN: 0097-8493.

Zhao, Hui (2006). "Implementing Surfaces in OpenGL". In: Waterloo: University of Waterloo, pp. 12–20.