KTH Computer Science
and Communication

Transparency with Deferred Shading

A protoype for rendering transparency with a Deferred Shader using Alpha-Blending

CHRISTIAN MAGNERFELT

Bachelor's Thesis at NADA
Supervisor: Mårten Björkman

Stockholm, Sweden 2012

Abstract

Transparency with Deferred Shading using Alpha-blending is a prototype that enables use of the Deferred Shadings technique to render transparent objects. The main feature of presented prototype is the ability to render transparent object in a post-pass using a front renderer with alpha-blending. A satisfactory performance of this designed method is demonstrated both for rendering speed as well for it being an artifact free for scenes with a low amount of lights.

Referat

Vi har skapat en prototyp som med Alpha-blending, tillsammans med Deferred Shading, kan rendera transparenta objekt. Vad som kännetecknar vår prototyp är att den bygger på front rendering som renderar alla transparenta objekt i en s.k. post iteration. Vi har visat att vår metod är artefakt lös samt inom acceptabel hastighet kan rendera scener med ett lågt antal ljuskällor.

Contents

Chapter 1

Introduction

Deferred shading has become a popular technique often used for development of main stream video-game applications. The deferred shading technique renders a scene in two passes, submitting only the geometry in the first and calculating lighting in the second. The first pass renders a screen space representation of the scene which is later saved in different buffers. These buffers are later used to calculate lighting in the second pass. Doing so reduces the amount of lighting calculations needed when the amount of light sources is high. However, handling transparency has been a major disadvantage of the deferred shading techniques since the introduction of the concept in computer graphics. In the first pass a visbility test is used to speed up and prevent rendering of occluded geometry. The test is performed to ensure that only lighting calculations relevant to the final image are included. However, due to graphic hardware limitations, only one pixel worth of information can be saved at the time of the visibilty test, therefore making it difficult to apply deferred shading technique for the rendering of transparent objects. Proper rendering of transparent objects dependents upon the visibility of each pixel. There are several ways to calculate this visibility. There are two most common techniques used for the final pixel color calculation: order-dependent and order-independent. The first one is based on the order of the rendered polygons. This method often requires that all the polygons are sorted in back-to-front order, thereby providing an artifact-free result. The second group of techniques is indifferent to the order of processed polygons. The purpose of our work was the development of an efficient technique for deferred rendering of transparent objects and solving the following problem:

How can we render transparent object using deferred shading?

Within the frame of this project, several techniques for rendering transparent objects were examined on their advantages and disadvantages. Below we suggest a review of previous studies on deferred shading as well as some of the practical solutions for application of this technique in transparency. We propose a prototype

of own developed technique for rendering of transparent objects. The built of our deferred shader is described along with actual integration, of the front renderer and deferred shader techniques, is explained in particular. We discuss the test results of our prototype in terms of performance and image quality and describe the model we used for this testing.

Chapter 2

Background and related work

There are several techniques used for handling transparency in a rasterizer. Below is an overview of some of these techniques.

A fragment holds just enough information to calculate a single pixel worth of color, in the framebuffer, which is later displayed on our screen. When dealing with multiple transparent fragments the final pixel color can be computed with the following equation recursively:

$$C_0 = a_0 c_0 \tag{2.1}$$

$$C_n = a_n c_n + (1 - a_n) C_n - 1 \tag{2.2}$$

Where a is the alpha value, C the color, Cn-1 the color of the previous fragment and Cn the final color. This technique is known as alpha-blending and was introduced by Porter and Duff[1] in 1984. The technique does not, however, yield a valid result unless the fragments are computed in a back-to-front order. To do so, each fragment needs to be sorted. Sorting the fragments is difficult, costly and does not always provide the satisfactory results.

The A-Buffer was introduced by Carpenter[2] in 1984. The A-Buffer is an algorithm implemented in hardware that stores transparent fragments for each pixel in a list. However, all of these fragments must be stored at the same time, which leads to an unbounded memory usage. In order for the algorithm to operate in a fixed memory space, dynamic memory allocation and management is required/mandatory. This is often associated with difficulties and, in most cases, costly to perform.

Techniques that avoid the sorting of polygons when rendering transparent objects is considered to be order-independent. Depth-peeling is one such technique and was introduced by Everitt[3] in 2001. It achieves order-independent transparency by peeling away one layer of transparent fragments at a time. By performing a depth test, the nearest visible fragment can be acquired. By doing additional n passes, n layers of fragments can be acquired. By doing n passes in this way may become costly with increased number of stacked fragments. There are, also, techniques which reduce the number of passes. In 2006, Liu et al[4] introduced a technique

that makes use of newer graphics hardware to render several layers in one pass. The technique uses a fragment program that sorts and writes multiple fragment colors and depth via multiple renderer targets[5]. The technique was later improved in 2008 by Bavoil and Meyers[6] using the min-max buffer to peel two layers at a time — one from the front and one from the back. This technique is known as Dual depth-peeling In 2009, Liu et al[7] improved the technique further by using bucket sorting to sort fragments by depth on the GPU.

Another technique that doesn't need sorting is screen-door transparency. With screen-door transparency, bitmasks are used to prevent certain pixels from being rasterized. This gives the illusion that the object is transparent. However, this technique fails to produce desirable image smoothness when compared to other techniques. Later, Mulder[8] (1998) suggested an improvement by optimizing the selection of stipple patterns. The screen-door transparency technique was further expanded by Eric et al[9] in 2010 and became known as stochastic transparency. This technique uses random sub-pixel stipple patterns, where each fragment of transparent geometry covers a random subset of pixel samples of a size proportional to alpha. This results in alpha-blended colors, on average. The disadvantage of this technique is, however, the noise.

One of the latest techniques for rendering transparent objects is adaptive transparency. This technique was introduced by Salvi et al[10] in 2011 and approximates closely to that of the A-Buffer. The difference is that it runs in a fixed memory space similar to that of the Z-Buffer. The Z-Buffer is responsible for performing visibility test of each fragment. Fragments that are occluded are discarded by the Z-Buffer and only visible fragment saved into the final framebuffer. The key point of the adaptive transparency algorithm is to adaptively compress visibility representation during rendering. However, results show that the technique can not truly run in a fixed memory space unless changes are made to the current graphics hardware.

In this work we will use the alpha-blending as the primary method to render transparent objects along with deferred shading for handling of nontransparent objects. Even though previously presented techniques may be faster and may produce a better result, alpha-blending is by far one of the simplest methods to implement.

Rendering using an off-screen buffer called the G-Buffer was first introduced in 1990 by Saito and Takahashi[11]. However, the contemporary graphic cards could not support the real-time application of this method. However, as the programmable pipeline was introduced along with Multiple Render Targets (MRT), it became possible to perform deferred shading in real-time.

There is one method to render transparent objects using deferred shading introduced by Kircher and Lawrance[12] 2009. Inferred lighting saves transparent objects as stipple patterns in G-Buffer.

Various popular video games have been developed using the deferred shading, i.e. S.T.A.L.K.E.R[13] and Tabula Rasa[14].

Chapter 3

Algorithm overview

Rendering transparent objects with deferred shading impose some problems as the depth-buffer used for rendering during deferred shading only supports one fragment at a time. In our work, we have chosen to use alpha-blending in a post pass using front rendering. Despite the flaws of alpha-blending, it is still very straightforward and is easy to implement into a deferred shader. In the next section we discuss basic functions of our algorithm. Blinn Phong[15] shading is a method often used for rendering the lighting in both the deferred shader and the front renderer and can used together with alpha-blending. A detailed overview of the front renderer algorithm is presented along with description of its' use of the basic functions. Finally, a more detailed description of deferred shading is presented along with comparison of this method to front rendering.

3.1 Basic functions

3.1.1 Alpha-blending

The alpha channel is used to store transparency in images. The data stored in the alpha channel ranges from 0.0 to 1.0. 1.0 indicates that the object is fully opaque, or does not let any light through, and where 0.0 indicates that the object is fully transparent. Common for most computer images is that the color and alpha value per pixel are often represented as RGBA, where RGB is the red, green and blue components and A the alpha component. Each of these is 1 byte in size, making up a total of 32 bits per pixel. We can use the alpha value to calculate the final color of a pixel. The first calculated fragment uses the function 2.1 which multiplies the alpha with the corresponding RGB color. The calculated value is then stored for the pixel in the frame buffer. All fragments are thereafter rendered using equation 2.2 , which takes the alpha of the current fragment multiplied with its RGB color. It then adds the previous rendered fragments' color multiplied with the 1-alpha of the current fragment. Thus, if the current fragment only has 0.75 alpha, only 0.25 of the previous fragment will be visible. In OpenGL, the blending equations 2.1 and 2.2 can be activated within the rasterizer by using the OpenGL commands :

```
// Activate Blending
glEnable (GL_BLEND);
// Set blend function
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

3.1.2 Blinn-Phong Shading

The lighting model we used for our deferred shader and front renderer is the Blinn-Phong lighting model. We chose this model because it is easy to implement for both the deferred shader and the front render, and is thereby also easier to compare. The model works as follows:

$$Half = Light + View/||Light + View|| \tag{3.1}$$

By using formula 3.1, we can calculate the half vector between the light and the view vector:

$$I = Diffuse * (Normal \cdot Light) + Specular * (Half \cdot Normal)^n \tag{3.2}$$

We can later use the half vector to calculate the amount of specular reflection at a given point using equation 3.2. If the angle between light and normal as well as eye and normal is equal then we have a perfect reflection and the maxium amount of specular color. n is the specular power.

3.2 Front Rendering

Front rendering is the traditional technique for rendering graphics. It sends data on geometry and lighting to the shader in order to calculate the final color for every pixel. However, for multiple lighting different approaches can be used. One way is to bind all lights into a single shader using a so-called uber shader. This does not scale as we have a fixed set of lights; one can, of course, approximate what light sources affect the object the most. However, doing so can be both difficult and expensive to perform. Also, there is the risk of maxing out the number of uniform bind-able values for the shader. The two other methods do scale and work similarly. The first method shown below renders all objects for each light. Successive passes then blend together the result. The second method does just the opposite and renders all lights for each object which are then also blended together.

```
For each light n
   for each object k
      calculate light n on object k

for each object k
   for each light n
      calculate light n on object k
```

However, there is a risk, with described techniques, to shade polygons and fragments that are not visible in the final image. The Z-buffer is used to perform the visibility testing to remove the non-visible fragments. and all geometry must be sent each time. Of course we can cull our geometry in our application but this is both very complex and very costly. Therefore these techniques has the time complexity of O (lights * objects) which rather poorly for scenes with many lights but works well when the number of lights is low.

3.3 Deferred Shading

Instead of rendering all lights for all objects, like in front rendering, one can render all scene geometry to an off screen buffer. Then, one can use this buffer to apply the lighting for all lights in screen space. This will allow to reduce the amount of wasted shading for fragments that are not visible in the final image.

The efficiency of Deferred shading is in its ability to effectively use the Z-Buffer to filter out any non-visible fragment. The depth value of the final visible fragments will be stored in the depth buffer.

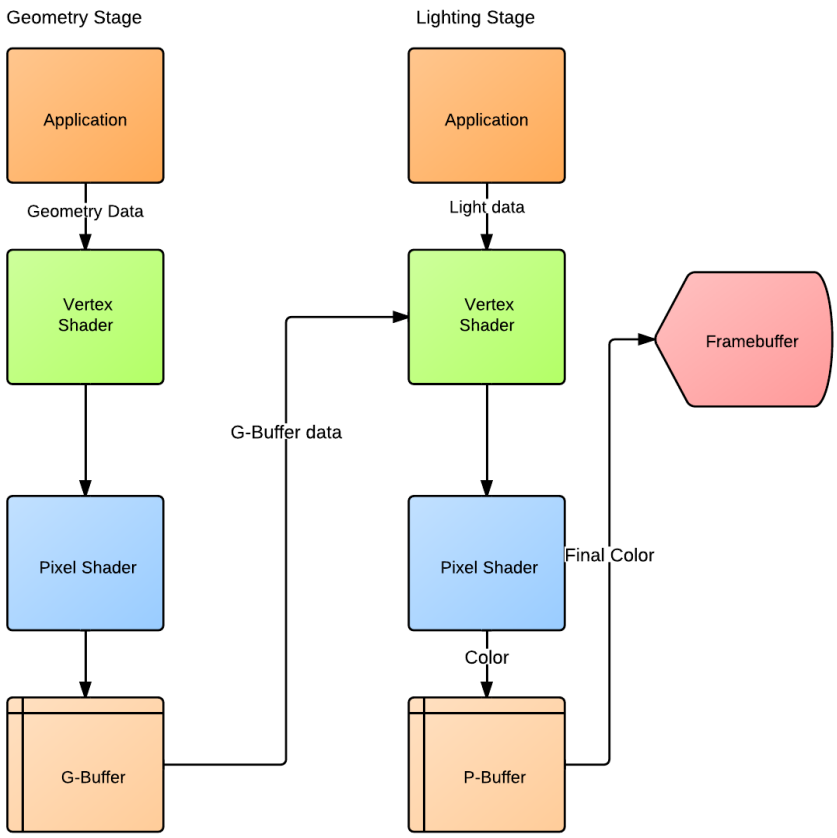
The buffer used by Deferred Shading is called the G-Buffer a.k.a. Geometric buffer. The G-Buffer stores relevant information about the geometry of the scene used for lighting calculations such as depth, normals and various material properties.

```
// First pass
For each object n
      render object n to G-Buffer

// Second pass
For each light n
      for each pixel k
            calculate light n on pixel k
```

Calculation of lighting in this manner may reduce the complexity to O (lights + objects). Note that constant O is higher due to the fact that each pixel has to be rendered at least once.

Figure 3.1: Geometry Stage: First geometry is sent to shader, then each material property is saved into the G-Buffer. Lighting Stage: The G-Buffer is later bound as texture in order to calculate the final color. The color is saved in the P-Buffer and later copied to the framebuffer



Chapter 4

Implementation

The deferred shading and the front rendering pipeline have already been implemented in various games and applications, the game S.T.A.L.K.E.R being on of them. Compared to game as S.T.A.L.K.E.R. our implementation uses newer graphics API for later generations of graphic hardware. However the G-Buffer have been simplified and only support one type of material which in this case is Blinn-Phong. Combining deferred shading with several types of materials as well as being compatible with older graphics hardware was considered to be outside the scope of this work. Our implementation does not have any major optimization apart from using newer API. Many of the optimization that could be done are either too difficult or take to long to be fitted inside this work. The whole application was built from scratch using C++, nvidia CG and OpenGL.

4.1 Deferred shading with Front Rendering

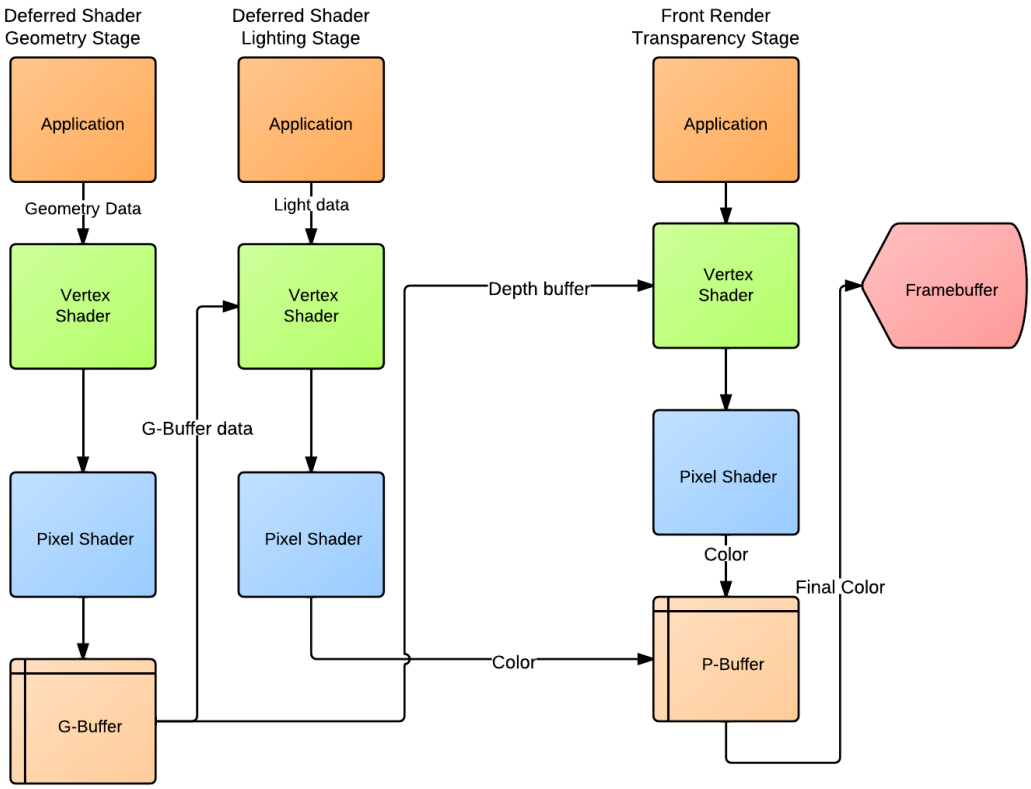
The front render is used to process transparent objects and fits well into the deferred shading pipeline. It renders all opaque objects first with the deferred shader and then renders the transparent objects on top using the front renderer. This is important as the depth buffer has to be filled with opaque objects first, to prevent rendering of non-visible transparent objects. When the front renderer is performed, the final picture can be rendered to the frame buffer for display.

The implementation has the following rendering stages:

1. Render all opaque geometry to the G-Buffer
2. Bind the G-Buffer as texture. For each light in the scene draw a full screen rectangle and calculate lighting at each pixel using the data from the G-Buffer. Save result in the P-Buffer.
3. Sort all transparent entities in back to front order.
4. Render all transparent geometry using the front renderer. Blend the result to the P-Buffer using the depth buffer to filter out any non-visible transparent geometry.

5. Copy P-Buffer to frame buffer.

Figure 4.1: After the final color from the Lighting Stage have been calculated, we bind the G-Buffer depth buffer as texture for the front renderer. The depth buffer is used to filter out any non-visible transparent geometry. The front render its result on top of the P-Buffer which is later copied to the framebuffer.



4.2 OpenGL Extentions

In order to simplify the deferred shading process, OpenGL FrameBufferObjects[16] are used for storing all information in the G-Buffer. This FrameBufferObject can easily be switched with the default frame buffer, as well as able to be rebound as texture. When the buffers are bound as texture they can be used in the same way as normal textures and can be used in lighting calculations. Note that FrameBufferObject extension is only available on graphic hardware that supports OpenGL 4.1 and does requires shader model 5.0. In order to accelerate the drawing process OpenGL VertexBufferObjects[17] are used to store all relevant information regarding the model attributes such as positions, normals, tangents and texture coordinates.

4.3 G-Buffer and P-Buffer

The G-Buffer which holds the geometry information in the deferred shader is very simple. It only holds the least amount of data to perform simple Blinn-Phong shading.

The first buffer is the depth-buffer which uses 3 bytes. The remaining buffers

Table 4.1: Structure of the G-buffer.

R8	G8	B8	A8	Target
Depth24	-	-	Unused	Depth
Diffuse R8	Diffuse G8	Diffuse B8	Unused	Color0
Normal X8	Normal Y8	Normal Z8	Unused	Color1
Specular R8	Specular G8	Specular B8	Shinyness8	Color2

are for the diffuse color, normals and specular color and shininess. Note that all buffers are in total of 4 bytes as required by the FrameBufferObject in order to work. The P-Buffer only hold R8 and G6 and B8 components which can later be copied to the framebuffer.

Table 4.2: Structure of the P-buffer.

R8	G8	B8	A8	Target
Color R8	Color G8	Color B8	Unused	Color3

4.4 Sorting

For sorting of transparent objects we use the built in stl::sort using Visual C++ 2010 with compiler version 16.00.40219.01. The average of a sort complexity is O(N

log N), where N = last – first[18]. Transparent objects are sorted by the distance from the camera to the object itself. It is preferable to sort each polygon for each entity in the scene in order to decrease the number of arte-facts. However, this may result in severe performance loss as each polygon would have to be translated each frame and reloaded into the video memory. The sort by object was chosen in this work due to simplicity of implementation.

Chapter 5

Results

Efficiency of our deferred shading renderer was compared with Front renderer.

5.1 Test Model

A simple cube model was used to test our deferred shader. The model consists of 36 vertices which make up for the total of 12 polygons. The rendering was performed 100 times both for opaque and transparent geometry. We used VertexBufferObjects to ensure the efficiency of time usage. The model properties including its texture are stored in video memory which improves the speed for loading these properties when the GPU need to render the model.

Figure 5.1: Diffuse color texture used for the test model

Figure 5.2: Normal map used for the test model

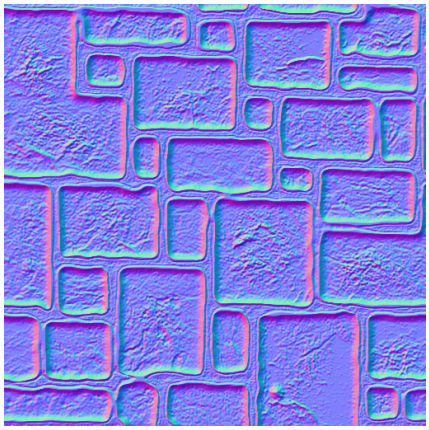


Table 5.1: Model properties.

Model	CUBE
Vertices	36
Polygons	12
Data	Vertex positions, normals, tangents and texture coordinates
Size in bytes	432 bytes

5.2 Performance

All tests were performed on a Intel Core i7-2600 3.40 Ghz CPU using a Radeon HD 6570 GPU on Windows 7 64 bit system. Resolutions tested were 1280 x 800 and 800 x 600, we choose these resolutions because we want to reflect resolution in modern applications.

The tables 5.2, 5.3, 5.4, 5.5 were used to calculate the performance diagram 5.3. The three columns to the right in each table represent the running time in application for each stage. Tests were done for 100 opaque (OP) CUBE models as well as 100 transparent (TR) CUBE models at both resolutions. In total there were 4 tests and as we can confirm from figure [5.3] deferred shading scale better for many lights. Even at 50 lights the deferred shader preforms well in 800 x 600 resolution. The front renderer does well with one light however as the number of light increases it directly losses it's ability to perform within the acceptable limits of a real time application which is around at least 30 Fps. From table 1 to 4 we note that for transparent objects the front renderer increases dramtically in running time as the number of light increases. This is most likely due to the front renderer doing matrix calculations in application.

Figure 5.3

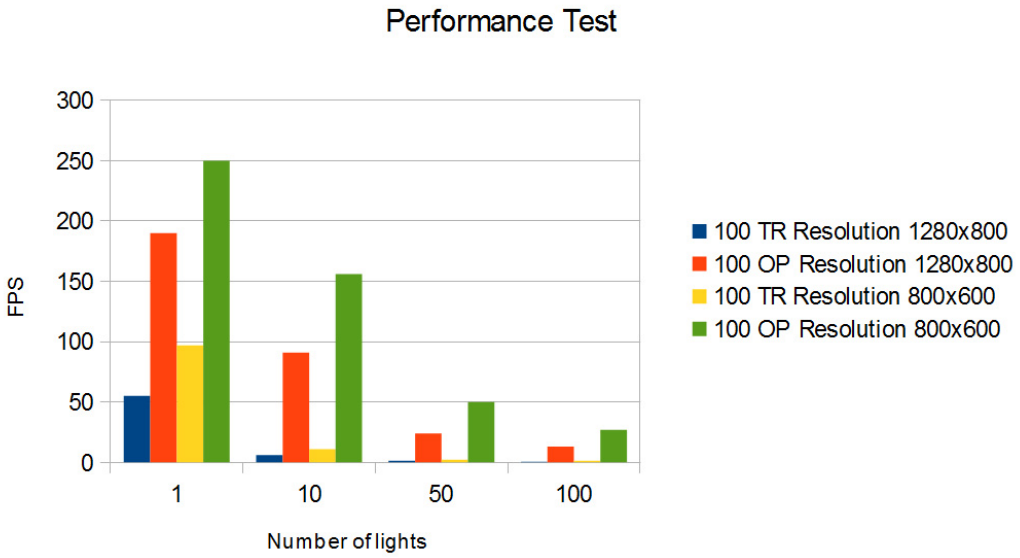


Table 5.2: 100 Transparent objects at 1280 x 800 resolution

Number of lights	FPS	Deferred Shader ms	Sorting ms	Front Renderer ms
1	55	0	1	2
10	6.18	0	1	10
50	1.25	0	1	53
100	0.6	0	1	88

Table 5.3: 100 Opaque objects at 1280 x 800 resultion

Number of lights	FPS	Deferred Shader ms	Sorting ms	Front Renderer ms
1	190	2	0	0
10	91	2	0	0
50	24	2	0	0
100	13	2	0	0

Table 5.4: 100 Transparent objects at 800 x 600 resolution

Number of lights	FPS	Deferred Shader ms	Sorting ms	Front Renderer ms
1	97	0	1	2
10	11	0	1	10
50	2.23	0	1	45
100	1.28	0	1	88

Table 5.5: 100 Opaque objects at 800 x 600 resolution

Number of lights	FPS	Deferred Shader ms	Sorting ms	Front Renderer ms
1	250	2	0	0
10	156	2	0	0
50	50	2	0	0
100	27	2	0	0

Figure 5.4: Buffer with diffuse color

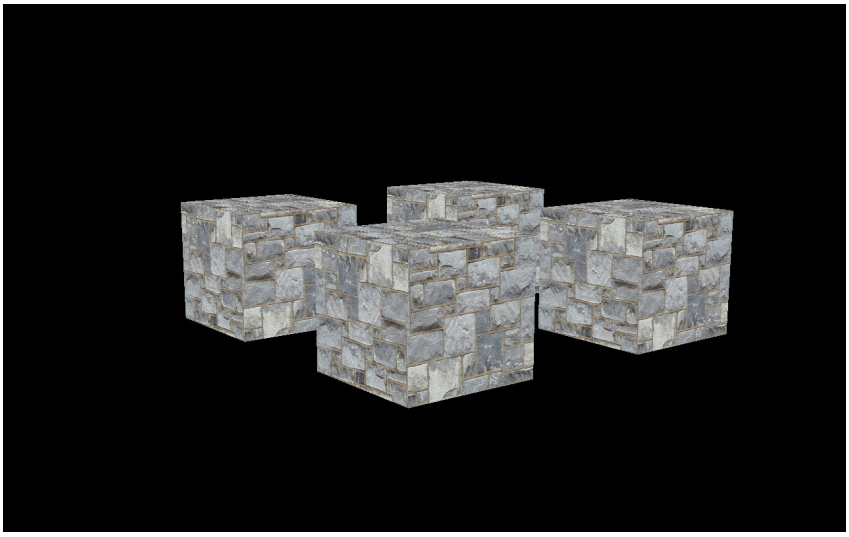


Figure 5.5: Buffer with normals

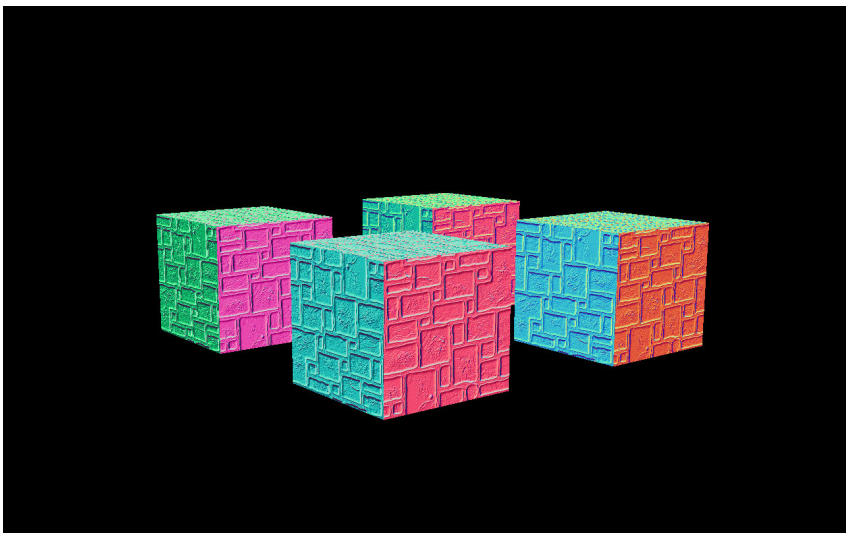


Figure 5.6: Buffer with specular color

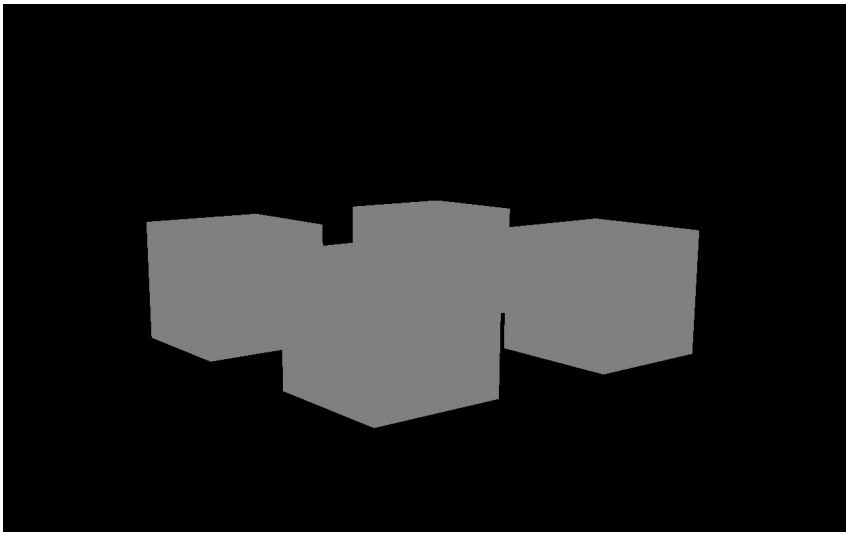


Figure 5.7: Depth buffer

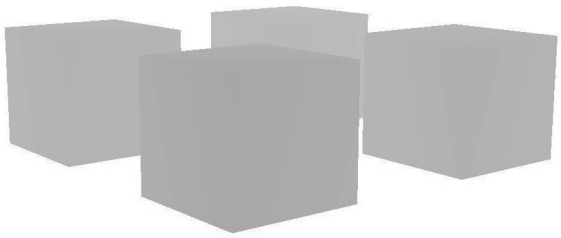


Figure 5.8: Final picture produced by the deferred shader using the previous 4 buffers

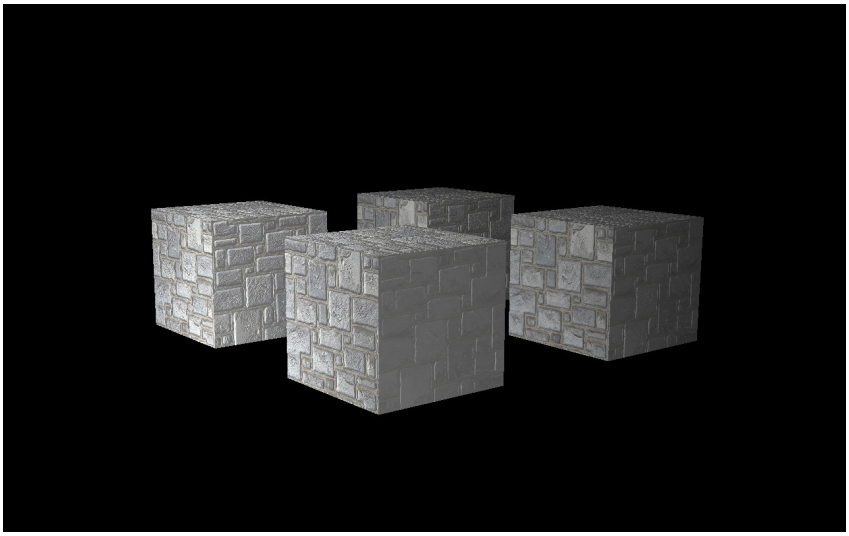
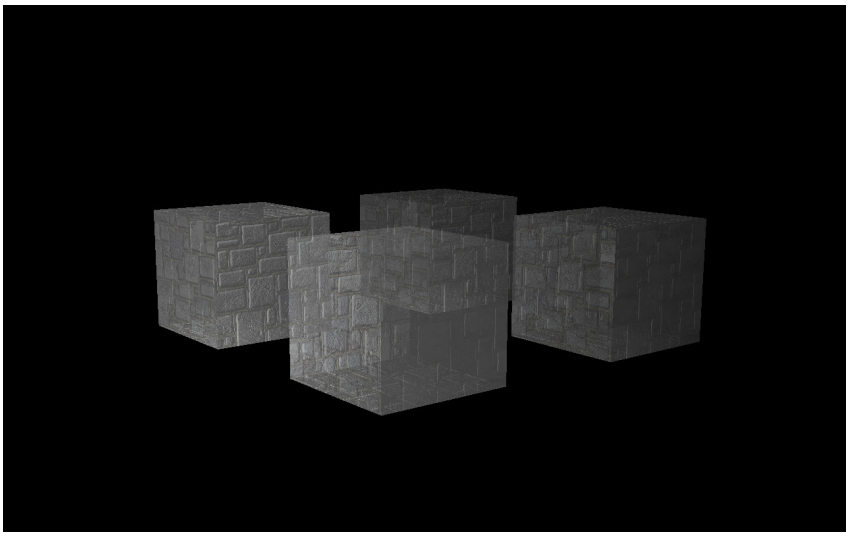


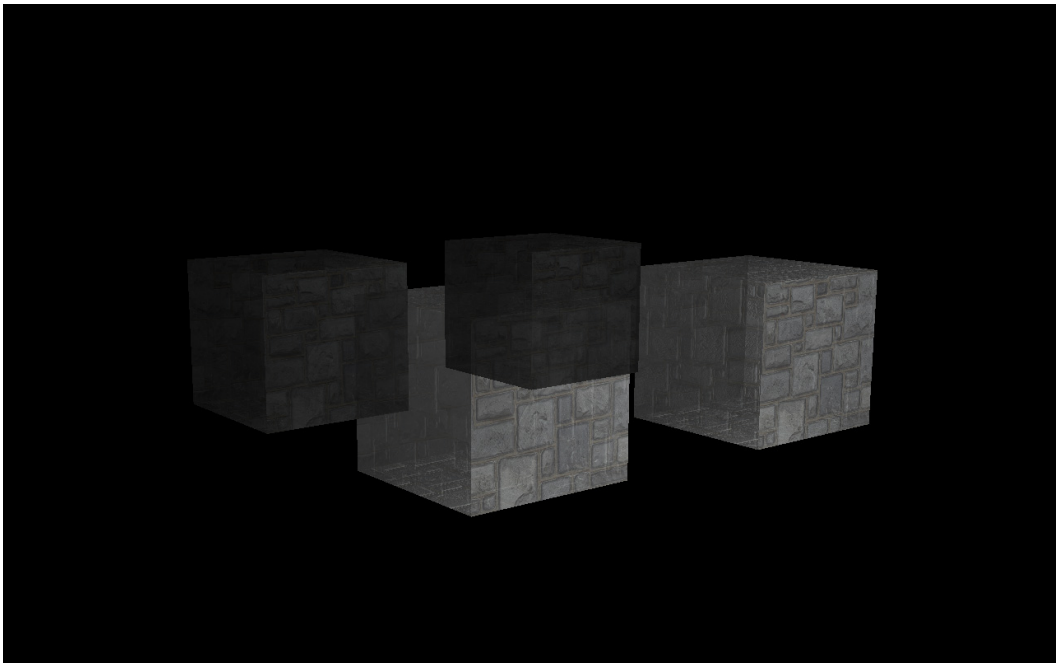
Figure 5.9: Same pictures but rendered transparent with a front renderer



5.3 Image Quality

If we take a look at a sample from the front renderer figure 5.10 we can clearly see that the result from using multiple lights doesn't look right. This has to do with the order for which objects are rendered. We render for each light all objects in the scene, however what this does is that we do not render in back to front order. If we had rendered all lights for each object and sorted all polygons, we could have avoided these artifacts.

Figure 5.10: Artifacts



Chapter 6

Conclusion and future work

As we increased the number of lights we effectively multiplied the number of objects that needs to be rendered. So given that we have 10 lights then each objects has to be rendered and blended ten times. This is not true for the deferred shader as noted by the results, for each added light we only draw one extra rectangle. We do lighting calculations at pixel level but the number extra calculations are far less than rendering all objects one more time.

We have shown that front rendering with alpha blending perform well enough to complement scenes rendered with deferred shading. We can use a low amount of transparent objects or reduce the number of lights that effect the transparent geometry. However in order to render transparent geometry in the same manner as deferred shading in term of performance newer techniques needs to be used. Techniques that do not use sorting but are order-independent.

6.1 References

1. Thomas Porter, Tom Duff, Compositing Digital Images, SIGGRAPH, New York, 1984, p253 - 259

2. Loren Carpenter, The A-buffer, an antialiased hidden surface method, SIGGRAPH, New York, 1984, p103-108

3. Cass Everitt, Interactive order-independent transparency, NVIDIA white paper, 2001, p3

4. Baoquan Liu, Li-Yi Wei, Ying-Qing Xu, Multi-layered deepth peeling via fragment sort, Microsoft Research Asia, 2006, p1

5. Microsoft, Multiple Render Targets (Direct3D 9), Microsoft Dev Center, viewed 2012.04.06

6. Louis Bavoil, Kevin Myers, Order Independent Tansparency with Dual Depth Peeling, Tech Rep, Nvidia, p3

7. Fang Liu et al, Effcient depth peeling via bucket sort, High Performance Graphics, New York, 2009

8. J.D. Mulder, F.C.A. Groen, J.J. Van Wijk, Pixel Masks for Screen-Door Transparency, Proc. Conf. Visualization, 1998, pp. 351-358

9. Eric Enderton, Erik Sintorn, Peter Shirley, David Leubke, Stochastic transparency, 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, New York, 2010, p1

10. Marco Salvi, Jeffersson Montgomery, Aaron Lefohn, Adaptive Transparency, HPG '11 Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, New York, 2011, p1

11. Takafumi Saito, Tokiichiro Takahashi, Compreshensible rendering of 3-D shapes, SIGGRAPH '90 Proceedings of the 17th annual conference on Computer graphics and interactive techniques, 1990, New York, 197-206

12. Scott Kircher, Alan Lawrance, Inferred lighting: fast dynamic lighting and shadows for opaque and translucent objects, Sandbox, Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games, 2009, New York, p39 - 45

13. Shishkovtsov, Deferred shading in s.t.a.l.k.e.r, In GPU Gems 2, Addison-Wesley, 2005, ch. 9, 143-166

14. Koonce, Rusty, Deferred Shading in Tabula Rasa, Hubert Nguyen GPU Gems 3, Addison-Wesley, 2007, p429-457

15. James F. Blinn, Models of light reflection for computer synthesized pictures, SIGGRAPH '77 Proceedings of the 4th annual conference on Computer graphics and interactive techniquesACM, New York, 1977. p1

16. FrameBuffer Object, OpenGL, viewed 2012.04.12

17. VertexBuffer Object, OpenGL, viewed 2012.04.12

18. MSDN , Visual C++ Standard Library <algorithm> sort, Microsoft, viewed 2012.04.11, cited on page 1

6.2 Statement of collaboration

The content of this document was written and produced by Christian Magnerfelt. All related software to this document was developed by Christian Magnerfelt.