Deferred Rendering Technique

MSc Computer Animation and Visual Effects 2011-12, NCCA, Thesis Report

Dionysios Toufexis
Abstract

The project thesis aims to demonstrate my own implementation of deferred Rendering as a technique of 3D graphics visualization using directional light. This technique is applied in games development to make the rendering of vast complex environments and lighting conditions more efficient. It is meant to explain in detail the steps taken in the development of the application of this particular technique using two different types of geometry. The first type is Infinite’s 3D Head Scan model[1] and the second type is Crytek’s Sponza[2][3] Model. In Chapter 1, I will introduce the reader to the concept of the deferred rendering technique, history, and disadvantages of using this particular technique. Then I will move to Chapter 2, and I will explain my implementation through figures and actual code. Finally in Chapter 3, I will discuss the future extensibility of my application with post process effects and my conclusion for this project.
About the project:

For this implementation I used C++, OpenGL 3.2 and GLSL 4.0. The development was done using the QtSDK framework with QtCreator 2.5 in Ubuntu Linux 12.04 using the Clang Compiler. I used the NGL library which is my course's standard library for developing 3D graphics applications. I tested my application in three separate computers. Two desktop computers and one laptop to make sure that it is easily portable and runs across different hardware on an optimal level. The minimum requirements for this application is: GPU with OpenGL 3.2 support and at least 512MB of VRAM. Nvidia series 8 and Ati series 6 will be able to handle the application fine. Although I could have used certain specific nvidia functions which would make the overall frame rate faster, I kept things simple and not exclusive, so it can be run on every system without specific hardware limitations. Below you will find the system specifications I used to test my application:

### 1st Test Machine
- **CPU**: Intel Core i7 3770K
- **GPU**: Nvidia GeForce GTX 680, 2GB DDR3
- **RAM**: 8GB DDR3 at 1600Mhz
- **OS**: Ubuntu 12.04

### 2nd Test Machine
- **CPU**: Intel Core 2 Quad 990
- **GPU**: Nvidia GeForce GT 8800, 512MB DDR2
- **RAM**: 4GB DDR2 at 400Mhz
- **OS**: Ubuntu 12.04

### 3rd Test Machine
- **Brand**: Dell XPS 17
- **CPU**: Intel Core i7 2700k
- **GPU**: Nvidia GeForce GT555m, 3GB DDR3
- **RAM**: 6GB DDR3 at 1333Mhz
- **OS**: Ubuntu 12.04
# Table of Contents

Abstract 1

About this project 2

Content 3

1. Introduction 4

1.1 Deferred Rendering Technique ........................................... 4
1.2 History of Deferred Rendering ........................................... 5
1.3 Deferred Rendering application ......................................... 5
1.4 Disadvantages ................................................................. 6

2. Implementation 7

2.1. The Class Diagram ........................................................ 7
2.2. Implementation ............................................................. 8

3. Extensibility 16

3.1. Post Process Effects ....................................................... 16
3.2 Future Extensibility ......................................................... 17

4. Conclusion 18

4.3 Conclusion ................................................................. 18
4.4. Acknowledgments ........................................................ 18
Sources ................................................................. 19
Bibliography ........................................................... 20
Introduction

1.1 Deferred Rendering Technique:

Currently, there are two rendering techniques used in the field of 3D graphics. While, Forward Rendering is the oldest and most standard technique used in computer graphics, an alternative technique, Deferred Rendering (or shading) has been applied in recent years, in games development in order to minimize computer system resources.

Forward Rendering involves rendering objects and applying the lighting calculation in the beginning of the pipeline. However, a major setback to this is that the same scene is rendered multiple times. To explain further, if an object in the scene is affected by multiple lights, it will be rendered x amount of times, based on the number of lights affecting the object in order to combine all the effects of each light. In other words, forward rendering does not cope well with multiple lights. Also another drawback is that lighting calculations will continue to occur even for the discarded pixels, (those pixels that are not part of an object in the scene), even through the blocked pixels won’t be shaded. Thus GPU resources will be waisted.

Deferred Rendering solves the problems described above by “splitting” the geometry and the lighting calculations to two different stages (opposed to Forward rendering that calculates the transformations and the lighting from the beginning until the end of the pipeline, the framebuffer).

In the first stage the vertex and fragment shader gathers the attributes required for the shading but no shading calculation actually takes place at the time. All transformations, normals, position, texture coordinates, depth (see figure 01) and generally information relevant to light shading are passed in a series of textures called the G-Buffer (thus Deferred rendering technique is consider to be an image space technique).

Figure 01: Example of a g-buffer. Diffuse, normal and depth textures.

GPU-GEMS 3, Tabula Rasa.

It is in the second stage that we do the actual light calculations (we still do the light calculation the same way as we normally do in forward rendering). The technique involves rendering the lights as a geometry, for directional lights we use a full screen quad, spheres for point lights and cones for spotlights. Then we use the textures stored in the g-buffer to calculate the color contribution of the light, per pixel.
1.2 History of Deferred Rendering

Deferred Rendering was introduced by Michael Deering at SIGGRAPH 1988. But as a concept it was conceived in 1990 by Takafumi Saito and Tokiichiro Takahashi in Siggraph introducing the g-buffer. Back then the name deferred was not used however. They introduced the geometry buffers which is up to today the fundamental way for implementing this technique. Although this concept was introduced in the early 90’s, a full scale implementation did not happen imminently. It took approximately twelve years to witness the first deferred shading implementation and only after another four years it gained increased popularity in the game development industry with titles like Crysis and Stalker (see figure 02) switching from the standard rendering technique to deferred.

![Figure 02: Deferred Lighting in Stalker. Stalker, 2007.](image)

1.3 Deferred Rendering Applications

As we have seen above, deferred rendering is mostly used in AAA game titles. The reason for that is that this particular technique is able to render complex world with dynamic lighting with much more ease and without creating a bottleneck in the system opposed to forwards rendering. Today’s big games use vast complex worlds with lot of light sources both outdoor and interior. Having so much dynamic lighting and a vast and complex world can strain the system’s resources. Instead of wasting computational cycles on geometry on pixels that are discarded we can light all the relevant pixels through the g-buffer maps. But it is important to understand that deferred rendering is just a mixture of image-based techniques. They all share a stage that they defer at some point in the pipeline. For example deferred lighting can be postponed for later but we can still render our geometry in forward style. Stalker (see Figure 02) for example is using only deferred lighting to calculate their light accumulation due to complex lighting conditions in the environment, while the rest of the engine still runs in forward rendering.
1.4 Disadvantages

We already explained the advantages of Deferred Rendering above, but it does have a few major disadvantages. Maybe the most important drawback is that the renderer cannot handle transparency within the deferred algorithm. Another major inconvenience is that it cannot handle hardware anti-aliasing (AA) correctly. This happens due to the splitting of the geometry and lighting calculations in two stages. It is possible to apply AA in the first stage, which is the geometry pass, but AA is mostly applied in the second stage, the light pass, when the lighting has been applied.

Transparency can be achieved by handling the transparent section of the scene later on. Anti-aliasing can be achieved by using edge detection algorithms (see figure 03), although GPU manufacturers have already come up with hardware anti aliasing like **FXAA, MSAA** and **TSAA**.

![Figure 03: Example of Edge detection. GPU-GEMS 2^{6}](image)
2.1. The Class Diagram

* The above class diagram does not provide classes that are outside the context of deferred rendering.
2.1. Implementation

We already explained how the deferred technique works in theory in page 4. Now it is time for the actual implementation. In order to understand this technique, one needs to understand how the framebuffer works and how we can create additional framebuffer slots to store textures. Thorough explanation of the framebuffer objects is not part of this report (as this is not a tutorial but a review of my own implementation) but the `FramebufferObjects` class will be a good explanatory of how framebuffer objects work.

**FramebufferObjects** Class:

The core of deferred rendering is the *g-buffer* (sort term for geometry). This is where we create our framebuffer object and generate the textures.

For implementing the first stage, called the **Geometry Pass**, we need first to create a default FBO and then generate the texture objects for the diffuse, world position and normals, depth and the final composite texture (see figure 04 and 05).

```
  glGenFramebuffersEXT(1, &m_fbo);
  glBindFramebufferEXT(GL_DRAW_FRAMEBUFFER_EXT, m_fbo);

  glGenTextures(1, &m_fboDiffuse);
  glBindTextureEXT(GL_TEXTURE_2D, m_fboDiffuse);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, WindowWidth, WindowHeight, 0, GL_RGB, GL_FLOAT, NULL);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

  Figure 04: the creation of the default FBO and the target binding.

  Figure 05: creation of the diffuse texture.
```

The creation of texture objects is very straightforward. I generate a texture called *m_fboDiffuse* and I bind it to a target of type `GL_TEXTURE_2D`. Then specify the color format of the texture, size, border, pixel data and the data type of the pixel data. Afterwards I specify a set of filters for the texture.

The procedure to generate the normals and world position textures is the same as the diffuse, with one minor change. The normals texture handles the interpolation of lighting into our objects in the scene, so it is wise to increase the channel bits. Instead of assigning the texture to be `GL_RGB` (which is 8bit by default) as I did with the diffuse texture, I assigned a `GL_RGB16F` (16 bit channel) so it can pass more information later on.

For the position texture I have assigned `GL_RGB32F` (32 bit channel) as this is going to map the entire scene. The depth texture is handled differently. The depth texture has its own parameter of type `GL_DEPTH32F` and takes no additional filters.

The last texture is the final composite texture, this will be used later on to output the final composite texture and it can be 8bit. Combining different number of channels is allowed in MRT but all must have the same number of bits[7].

So far I have allocated the buffers needed for the MRT (*multi render targets*) but I did not attach them to the FBO.
At this moment the buffers are empty. In order to start writing in the textures they need to be attached to an FBO (see figure 06). An FBO can have multiple texture attachments.

```
    glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, m_fboDiffuse, 0);
```

![Figure 06: binding the textures to the fbo.](image)

For binding a texture buffer to an FBO, I specify the target to be of type `GL_DRAW_FRAMEBUFFER` or `GL_FRAMEBUFFER` and I assign a color attachment to it (attachments are logical attachments of the framebuffer). Then I specify the type of texture to be expected and the name of the texture object to be attached. Binding the additional textures to the FBO follows the same tactic with the exception of the depth texture, as this needs its own attachment of type `GL_DEPTH_ATTACHMENT`.

**GeometryPass** Class:

Now that the g-buffer is ready I can start the MRT. This is done through the geometry pass shaders.

**Vertex Shader:**

```
TexCoord          = VertexTexCoord;
Normal             = (ViewMatrix * vec4(VertexNormal,0.0)).xyz;
WorldPosition      = (ViewMatrix * vec4(VertexPosition,1.0)).xyz;
gl_Position        = ModelViewProjectionMatrix * vec4(VertexPosition,1.0);
```

The vertex shader is simple. It does the matrix calculations and passess the attributes to the fragment shader.

**Fragment Shader:**

```
FragWorldPosition  = WorldPosition;
Albedo            = texture(DiffuseMap, TexCoord).xyz;
FragmentNormal    = normalize(Normal);
FragTexCoord      = vec3(TexCoord,0.0);
```

The fragment shader is responsible of doing the MRT. In forward rendering we normally output a single vector and then send it to the framebuffer for screening. Instead, I am sending a series of outputs in to the corresponding texture buffers. This is done by creating a buffer array (see figure 07) with the color attachment we assigned in the texture buffers.
Figure 07: The buffer array.

`glDrawBuffers` defines an array of buffers where outputs from the fragment shader will be written. That way if the fragment shader writes a value to one or more user defined output variables, then the value of each variable will be written into the buffer specified at a location within `Buffers`, pointing to the attachment assigned to the user defined output[8].

Before we start rendering we need to enable the framebuffer for writing and clear the framebuffer object from any previous data. After that, we do the camera transformations and render the mesh.

So far I have created the framebuffer object and manage to do MRT (see figure 08).

![Figure 08: Geometry Pass output, word position, normals and diffuse textures.](image)

Light shading will be implemented in the next stage called, the **Light Pass**.
Now that the geometry pass is ready and I have acquired all the relevant textures for the shading calculations, I can proceed with lighting the scene. This is done through the **Light Pass** class. Light pass is responsible for sampling each texture we previously created pixel by pixel and is also responsible for acquiring all the pixel values from the different textures and calculate the lighting\(^9\). In forward rendering this was simple to do. The vertex shader was responsible for providing the object coordinates in the world space and the fragment shader was responsible for shading each pixel.

But in deferred we have textures rather than actual geometry. Geometry is already processed and the visible geometry is known at each pixel location. Sampling each pixel is done by creating a full screen quad that is equal to the window coordinates. Why a **quad** and not a **sphere** or another type of geometry? As we mentioned in page 4, lights are represented as geometry.

In computer graphics a point light can be represented as a point in space (source) that emits light (vectors) to all directions. But in real life when light is emitted from a source it loses power over time and space. In computer graphics the vectors can continue indefinitely. To counter this, we simply create a sphere geometry and use it as a means of calculating the attenuation of those vectors (see figure 09).

![Image](image1.png)

*Figure 09: A representation of a point light in computer graphics.*

Pixels that are outside of the boundaries of the sphere will not be shaded. The same applies for spot lights. Spot lights have an angle and a designated direction in space and we represent that as cone geometry (see figure 10).
My implementation consists of one directional light. In the point of view of the light, the directional light emits vectors in the entire scene, so every pixel in the screen is shaded. That is why using a full screen quad is the best approach (see figure 11).

The other types of lighting, such as point light and spot light share some attributes and methodology. This is why the same approach as the directional light can be use also for those.
Now that we covered how light pass work lets review the implementation.

**Light Pass** Class:

I start by activating the texture coordinate units, and bind the textures so we can sample from the textures in the fragment shader. *(see figure 11)*.

```glsl
glActiveTextureARB(GL_TEXTURE0);
gbBindTextureEXT (GL_TEXTURE_2D, FBO.m_fboPosition);
glActiveTextureARB(GL_TEXTURE1);
gbBindTextureEXT (GL_TEXTURE_2D, FBO.m_fboDiffuse);
glActiveTextureARB(GL_TEXTURE2);
gbBindTextureEXT (GL_TEXTURE_2D, FBO.m_fboNormal);
```

*Figure 11: activating the texture units and bind them to the FBO.*

Afterwards I enable the blend map for drawing. Everything will be drawn in the fourth color attachment which is our final map.

```glsl
glDrawBuffer (GL_COLOR_ATTACHMENT4);
```

Now we can start doing the lighting calculation by enabling the light pass shader. The vertex shader for the light pass is identical with the one in geometry pass *(see figure 12)*. The only difference is that, for directional lighting, we specify the `ModelViewProjection` matrix to be the *identity* matrix (we do that so one world coordinate unit will be equal to one screen pixel). In this way quad vertices’s will not be affected and after the transformation happens we will have a quad according to the window width and height coordinates.

```
TexCoord = VertexTexCoord;
Normal = (ViewMatrix * vec4(VerteXNormal,0.0)).xyz;
WorldPosition = (ViewMatrix * vec4(Position, 1.0)).xyz;
gl_Position = ModelViewProjectionMatrix * vec4(Position, 1.0);
```

*Figure 12: the light pass vertex shader*

In the fragment shader I implement the BRDF and perform the lighting calculations. The main function is responsible for sampling the framebuffer textures and outputting the final color.

```
vec2 TexCoord = gl_FragCoord.xy / ScreenSize.xy;
vec3 WorldPosition = texture(PositionMap, TexCoord).xyz;
vec3 Color = texture(ColorMap, TexCoord).xyz;
vec3 Normal = texture(NormalMap, TexCoord).xyz;
vec4 aoFactor = texture(AOTex, TexCoord);
FragColor = vec4(Color, 1.0) * DirectionalLight(WorldPosition, Normal);
```
In order to pass light control from the light pass shader to the gl context we use uniforms. With uniforms we can manipulate the attributes of the shader from inside the application. To do that we need to set the directional light first. This is done within the LightSetup Class (see figure 13):

```cpp
4 | DirectionalLight::DirectionalLight()
5 | {
6 |   Color = ngl::Colour(0.0f, 0.0f, 0.0f);
7 |   Direction = ngl::Vec3(0.0f, 0.0f, 0.0f);
8 |   AmbientIntensity = 0.0f;
9 |   DiffuseIntensity = 0.0f;
10 | }

11 | void DirectionalLight::LightControl()
12 | {
13 |   Color = ngl::Colour(1.0f, 0.9f, 0.8f);
14 |   Direction = ngl::Vec3(10.0f, -2.0f, 0.0f);
15 |   AmbientIntensity = 0.6f;
16 |   DiffuseIntensity = 0.1f;
17 | }
```

*Figure 13: the LightSetup Class*

The implementation is almost done. The final part is called Final Pass. Final Pass is responsible for outputting the final result. In order to start having output, we bind the source framebuffer to read operation and bind the destination for drawing (see figure 14).

```cpp
glBindFramebufferEXT(GL_DRAW_FRAMEBUFFER, 0);
glBindFramebufferEXT(GL_READ_FRAMEBUFFER, FBO.getFrameBufferObject());
```

*Figure 14:*

Finally we specify the color buffer to be our composite attachment and we transfer the pixels to the default framebuffer.

```cpp
glReadBuffer(GL_COLOR_ATTACHMENT4);
glBlitFramebuffer(0, 0, m_WindowWidth, m_WindowHeight,
                 0, 0, m_WindowWidth, m_WindowHeight, GL_COLOR_BUFFER_BIT, GL_LINEAR);
```

*glBlitFramebuffer* will simple copy a set of pixels from the read FBO to the default framebuffer. In our case the source is the forth color attachment (*blit operations can only read from the color buffers that were specified in glDrawBuffers*) that is attached to the final texture. The *glBlitFramebuffer* command also allows to specify the textures location and size. That way we can produce multiple blits with different coordinates and output all our texture buffers in the screen.

The final output can be seen in *page 15 (figure 15,16).*
Figure 15: the final output for the human model.

Figure 16: the final output for the sponza model.

Before we display the buffer we can apply certain techniques that have various effects in the overall appearance of the final output. Those techniques are called Post Process Effects. Post process effects are applied in the end of the pipeline through the fragment shader. Examples of those effects are Full Screen Anti-Aliasing (FSAA), Bloom Lighting, Depth of Field (DoF), Field of View (FoV), Screen Space Ambient Occlusion (SSA) or Pre Baked Ambient Occlusion (PBAO) and many more. As seen in page 6, one of the major disadvantages on using the deferred rendering is the inability to apply hardware anti-aliasing out of the box. Game Development companies are using post process effects to overcome this particular problem by applying Gaussian filters or edge detection algorithms. Further more, with the increase need for fast anti-aliasing, GPU vendors like AMD and Nvidia are providing their own hardware AA solutions that run more efficient and accurate without straining the GPU resources.

![Figure 17: Bloom Lighting Effect. Far Cry. Crytek.](image)

Deferred Rendering maybe is the best example to apply such effects. Having the framebuffer object ready, we can apply more texture attachments to the FBO and start building the post process effects from there. In section 3.2, I will write for the future extendability of my application and what post process effects I want to add.
3.2 Future Extensibility

Most of the time during my implementation was distributed in the actual differed renderer engine in order to have a solid framework and expand it further in the near future. Currently my implementation supports rendering a scene with directional light and has a semi support for PBAO (the extensibility values don’t work quite well yet). Although PBAO is more of a trick rather than an actual technique used by graphics engine during 2007-2009. With the increase in hardware capabilities, it is possible now to add real-time occlusion effects as an image based technique (SSAO, HDAO, etc). SSAO implementation (see figure 18) is a post process effect and it one the first of addition will add to my renderer. A nice and thorough example can be found in gamerendering[14] website. Furthermore, my priority is to add additional types of light and shadows. Jon Macey[15] provides thorough examples on how to implement both shadow maps and additional types of light. Also a more advance types of BRDF must be implemented to raise the visual quality for interior scenes.

Figure 18: By enabling SSO the visual quality is greatly increased. Crysis 2
4.1 Conclusion

The overall application design is well written but it can become more efficient by developing methods that can pass extensive control on how the framebuffer object behave. For example instead of assigning the framebuffer object in a GL_FRAMEBUFFER target we can explicitly set two methods for activating the framebuffer for drawing operation by binding the FBO to GL_DRAW_FRAMEBUFFER and activate the FBO for read operation by binding it to a GL_READ_FRAMEBUFFER target. Whereas GL_FRAMEBUFFER will bind the FBO for both drawing and read operation without giving control to the programmer to assign an operation manually.

Overall this project was successful on creating a solid engine, that future extensions, can be implemented without modifying the engine. The engine is designed in such a way that can receive additional geometry in the first stage (geometry pass) with no additional coding (incorporating Jon Macey’s sponza model application was a solid example), as well as, implementing additional techniques with much more ease.

4.2 Acknowledgments

I would like to specially thank my supervisor Mr. Jon Macey for his constant guidance and support throughout the development of my project. His contribution to the development of the Sponza model import application, proved a valuable addition that enabled me to greatly enhance the visual aesthetic of the final output, as well as, test how versatile my application is on importing additional complex geometry.
4. Bibliography

Footnotes

[1] Lee P.S. *Infinite*. (September 26, 2010). **Infinite, 3D Head Scan**.[Web Site]  
Available at: http://www.ir-ltd.net/infinite-3d-head-scan-released (Accessed 15 October 2012)


Address: http://nccastaff.bournemouth.ac.uk/jmacey/GraphicsLib/Demos/index.html  
(Accessed 6 December 2012)

Available at: http://ephenationopengl.blogspot.gr/2012/01/setting-up-deferred-shader.html (Accessed 2 October 2012)


[8] OpenGL. *Documentation*. **glDrawBuffers**.[Online Documentation]  
Available at: http://www.opengl.org/sdk/docs/man/xhtml/glDrawBuffers.xml  
(Accessed 3 December 2012)

Available at: http://www.beyond3d.com/content/articles/19/3 (Accessed 6 October 2012)


Available at: http://www.opengl.org/wiki/Framebuffer_Object (Accessed 14 November 2012)


(Accessed 4 December 2012)


Address: http://nccastaff.bournemouth.ac.uk/jmacey/GraphicsLib/Demos/index.html

(Accessed 6 December 2012)


---

**Bibliography**


- Hargreaves, S. *ClimaxTM Deferred Shading.* Game Developers Conference. [Slides].


- Guerrilla. *Developer Conference. (July 07). KillZone.* [Slides].

• Game Rendering. (November 01, 2008). **Deferred Lighting.** [Web Article].

• Madison, W. Human Head Studios. (August 10, 2008). **Deferred rendering.** [Blog Article].

• Calver, D. **Beyond3D.** Jul 31st, 2003. **Photo-realistic Deferred Lighting.** [Web Article].
  Address: [http://www.beyond3d.com/content/articles/19/](http://www.beyond3d.com/content/articles/19/) (Accessed 1 November 2012)

• Pensjö, L. **Ephenation OpenGL.** (Jan 7, 2012). **Setting up a deferred shader.** [Web Article].
  Address: [http://ephenationopengl.blogspot.gr/2012/01/setting-up-deferred-shader.html](http://ephenationopengl.blogspot.gr/2012/01/setting-up-deferred-shader.html)