Advanced Deferred Rendering Techniques

NCCA, Thesis Portfolio
Peter Smith

August 2011

Abstract

The following paper catalogues the improvements made to a Deferred Renderer created for an earlier NCCA project. It covers advanced techniques in post processing, specifically referring to screen-space anti-aliasing. The work presented surrounds this second iteration of the renderer and the relevant improvements to features and system architecture.

Contents

List of Figures

Chapter 1

Introduction

The following work will attempt to explain the need for, and use of advanced image processing techniques in a deferred renderer. As such, the fundamentals of what a deferred shading pipeline is will not be covered in great depth; only a brief explanation will be offered. For more information on the basis of this renderer, see Smith, 2011 [1].

1.1 Deferred Renderer: A Summary

Deferred rendering [1] is a method of real-time rendering that has gained popularity in the games industry in recent years. The traditional method of forward rendering shades a scene on a per-object basis, limiting the amount of lights able to exist in a scene. This is because per-object lighting calculations can quickly become computationally expensive in large scenes. The main idea of deferred rendering is to move the lighting further down in the pipeline. This is done by rendering the entire scene once, to a frame buffer object (FBO). An FBO can be defined as a container for several pixel arrays of the same dimensions. They can have several textures drawn into their colour attachments, mimicking the behaviour of the screen or default frame buffer. [2]

The scene is rendered to an FBO called the geometry buffer (G-buffer). Various different types of information can be stored in the red, green, blue and alpha (RGBA) channels of these textures. Typically however, the key data is the lighting information. In the G-buffer the albedo (diffuse) colour contribution of the rendered surfaces, the vertex normals and the vertex position are stored. In addition to this specular information and other material attributes can be held. Figure 1.1 shows the g-buffer layout of the current version of the renderer:

With the G-buffer stored, geometry is rasterized which approximates the shape of our lights.[3] Whilst in the fragment shader for each of these lights, the G-buffer textures are sampled and the scene is lit on a per-pixel basis. The lighting calculations vary

[1] sometimes called deferred shading or lighting

[2] at the time of writing, the typical number of colour attachments available is four

[3] A cube for a point light, a cone for a spot light etc

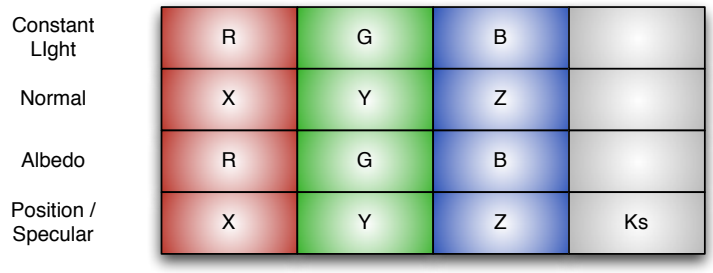

Figure 1.1: What Makes up The g-buffer

depending on light type but the outcome is similar; a new texture is output based on a blend of the scene and the lights. See Figure 1.2.

When the light accumulation is complete post process operations on the final lit image can be started.

1.2 Post Processing and Advanced Techniques

Post processing is a step that occurs at the end of both the forward and deferred rendering pipelines. It is in this stage that the 'final' lit image texture is sampled using a wide array of shaders. These shaders apply various aesthetic effects by drawing textures onto full screen quads. The process is sometimes referred to as applying full-screen effects. The basis of this idea is that a new pixel value can be calculated based on pixel values that already exist in texture memory. The pseudo-code for a simple grayscale effect might look something like this:

```
src = Sample the source pixel
grey = src.r * 0.21 + src.g * 0.71 + src * 0.07
output = Vec4(grey)
```

In general, a post-process effect will read in a source texture, manipulate samples from this map, and output a new texture map with the affected pixel values. In some cases however, effects need to sample many maps to attain the appropriate information. For example, a motion blur effect would require both colour and velocity maps as input. Some examples of post-process effects are:

- Bloom
- Radial Blur
- Gaussian Blur
- Color Grading (Greyscale, Sepia)

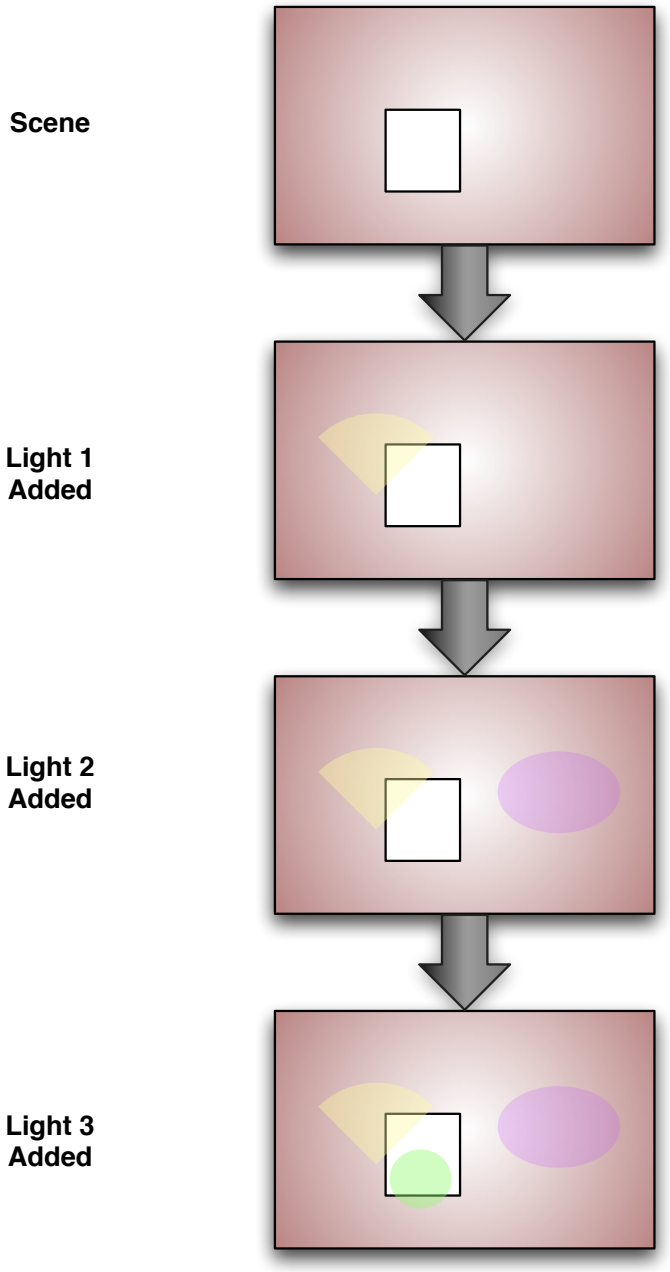

Figure 1.2: Symbolic representation of light accumulation

- Depth of Field
- Motion Blur
- Night Vision Filter

The use of effects like these has seen a rise in the variety of aesthetics found in modern computer games.

In a deferred renderer however, these effects have much more practical implications. There are two key areas of the deferred pipeline that are not hardware supported as they in forward rendering: anti-aliasing and transparency. In this implementation, a solution for anti-aliasing proposed by nVidia was explored [4].This solution uses comparisons in depth and normals to detect edges used in anti-aliasing.

The remainder of this paper will explain the implementation of three key areas of a deferred renderer: application architecture, post-process effects and anti-aliasing.

Chapter 2

Implementation Review

The following chapter assesses the implementation of improved application architecture, in comparison to the first version of the renderer. It will also explore how several full-screen effects can be implemented and their results. Finally, a solution to anti-aliasing will be explicated.

2.1 Application Architecture

Earlier iterations of the renderer saw a reliance on and instance of the FrameBufferObject class to control the input and output textures of the G-buffer. This implementation was not generic and did not allow a very wide-spread application control of textures; it was too restrictive. It was this factor that motivated a separation of texture logic from FBO logic.

This means by having an application-wide engine for all textures, users have to be more explicit when binding and altering them. Naming textures enables higher verbosity in the code, whilst also keeping the flow of control of texture units transparent. In the context of a deferred renderer, this is a major advantage as the key element of the pipeline is the textures.

Another goal of the texture management system is to facilitate a simpler pipeline for post-process effects. A single effect can be identified as having the following properties.

- Input Texture(s)
- Shader
- Output Texture(s)
- Name

So, in order to encapsulate an effects behaviour, methods were implemented to allow for drawing preparation, input/output configuration and execution. This was then wrapped into a Post-Processor in order to manage the different effects in the pipeline.

The advantage of this structure is that, in future iterations of the renderer, effects options will be read from configuration files to allow ease of use. This architecture, in turn will allow for automatic generation of textures on a per-effect basis; removing user-responsibility for a large part of the post-process pipeline.

This architecture is also echoed in the renderers Lighting system. It was designed with user-configuration in mind. A utility class that manages all the lights in the scene was implemented to allow for simple configuration of each type of light from a scene description file.

2.2 Full-Screen Effects

One of the most fundamental post-process effects is *Gaussian Blur* . Not only can it be applied easily for full-screen blur, but it can be used in a variety of other effects. The method for applying screen-space Gaussian Blur is one of weighted averaging. Firstly, pixels on the left and right are sampled and a proportion of these values is blended into the current pixel and written to a new texture. This process is then repeated vertically. These two outputs can then be combined to apply full-screen blur. Code for horizontal blur is listed in Appendix A.1.

Another widely used post-process technique is bloom. Lionhead and Bethesda have used this technique widely in their fantasy games as shown in figure 2.1. The reason for this is that it applies a certain ethereal glow to the brightest areas of the image.

There are different ways to implement this technique. One method is to fill a value in the G-buffer to indicate to the post-process engine to what extent bloom should be applied. This method of bloom allows a very high level of control over what parts of the scene will glow. Another method is to sample the texture that needs bloom applied and take the brightest pixels based on a pre-defined tolerance. Put simply, the brightest pixels are sampled and enhanced. This is not done simply by brightening them, but by outputting a bloom-map, applying Gaussian Blur to it and then blending it back into the scene. Figures 2.2 and 2.3 show these the scene with and without bloom.

In addition to blur and bloom a simple implementation of depth of field was implemented. Simulating depth of field in screen space is a process of sampling depth and blurring the areas not 'in focus'. This is a simplified simulation of an effect created by the focal length of cameras. Despite being a rather naive implementation, the effect achieved is still aesthetically pleasing; figures 2.4 and 2.5 show this.

2.3 Anti-Aliasing

Anti Aliasing refers to the smoothing of edges to counter-balance inherent coarseness of shader sampling rates. One of the major drawbacks of using deferred rendering is the lack of hardware support for this feature. This essentially means that the current

Figure 2.1: Examples of bloom. Top: Bethesdas "Elder Scrolls 4: Oblivion". Bottom: Lionheads"Fable 3"

hardware design and OpenGL implementation does not support user screen-space anti-aliasing. It cannot be used when rendering to an FBO. A solution for this is presented by Nvidia in GPU Gems [4]. This solution was adopted and integrated into the renderer.

The premise of the algorithm is that, by sampling the depth and the normal of a pixel in relation to its surrounding pixels, a weighting can be obtained for the amount of anti-aliasing blur it should receive. For this to work, the normal and position buffers can be sampled from the g-buffer and used in the following way:

- Calculate each angle cosine between current and 4 surrounding pixels
- Use these values to calculate a weighting value
- Calculate the difference in depth of the current pixel and 4 surrounding pixels
- Use these depth values to calculate a second weighting

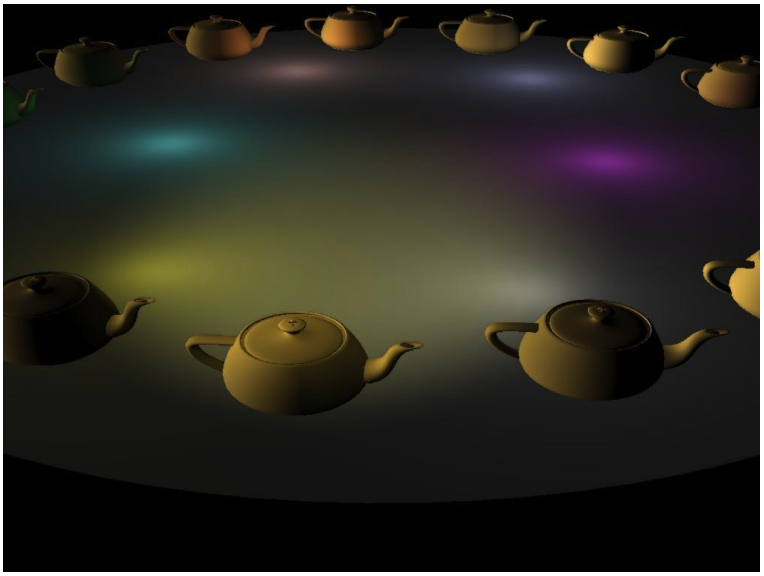

Figure 2.2: Scene Without Bloom

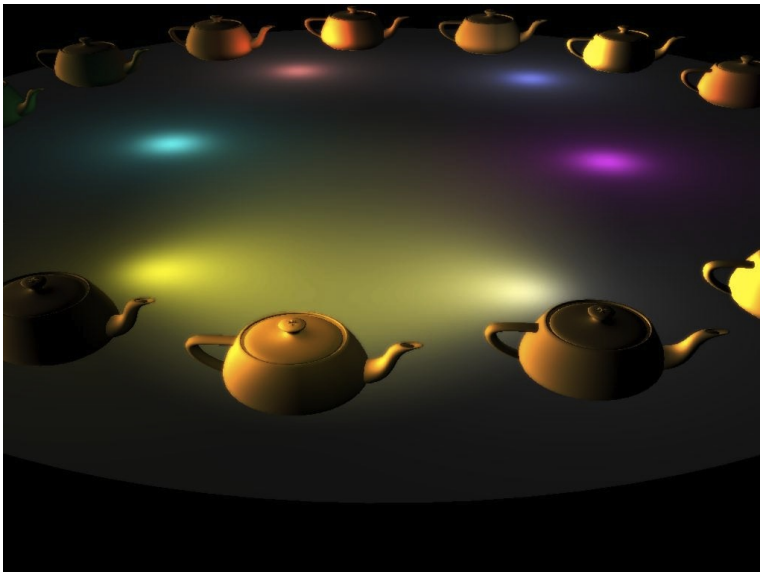

Figure 2.3: Scene with Bloom

- Sample the 4 surrounding pixels scaled by the coefficients and make the current pixel an average of these

Figure 2.6 shows the edge detection in action.

As described above these edge are then used to smooth the colour of the current pixel. Appendix A.2 shows the colour-blending portion of the shader.

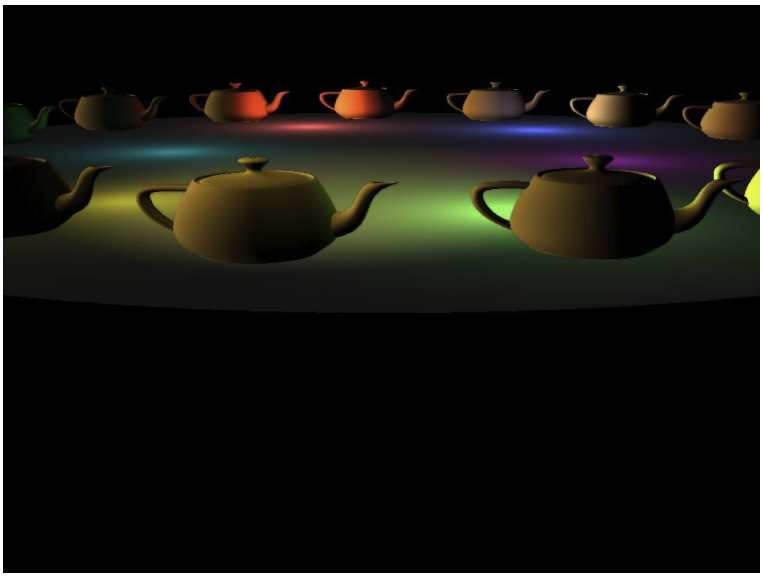

Figure 2.4: Scene Without Depth of Field

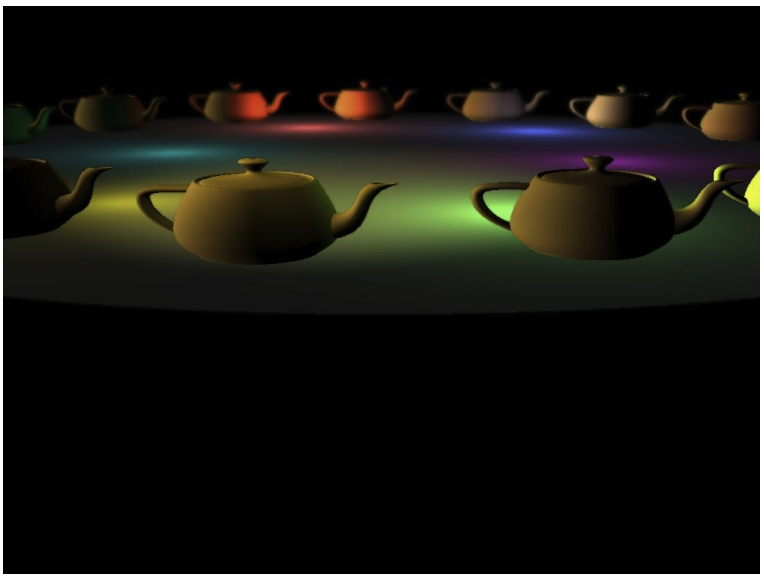

Figure 2.5: Scene with Depth of Field

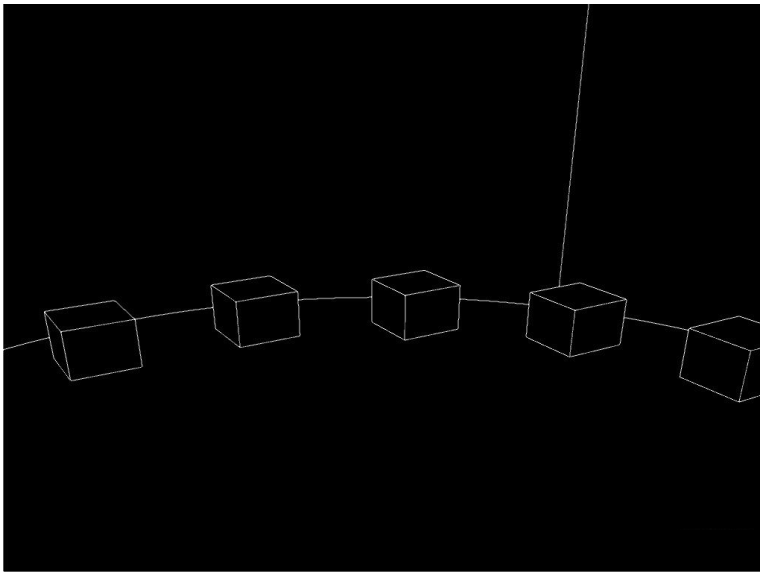

Figure 2.6: Edge Detection for AA

Chapter 3

Conclusion and Future Work

3.1 Future Work

The work explicated above summarises a short list of changes and improvements made as a second iteration of the earlier renderer. It is a feature of this portfolio, and as such, can be expanded upon and improved. For example, in future iterations of the renderer the problem of transparency will be handled which is likely to involve performing a final pass to re-render transparent objects.

In addition to this the next version of the renderer will be written to accommodate a complete pipeline. The ability to communicate with the renderer in a generic way is an important feature that will allows future abstraction and extension. The aim is to allow a general description of geometry and scene layout to be supplied that will allow users to simply specify their mesh, texture and lighting details. This will, in turn allow the renderer to handle the rest of the process. In addition to these higher level features there are some low-level optimisations that can be made. An example of this would be re-structuring the G-buffer to allow more lighting information to be stored. This could be achieved by encoding position into one depth value is one examples of this.

3.2 Conclusion

Overall, the outcome of this second iteration of the renderer was successful. The overhaul of design allowed for the post-process stage to be implemented more easily. The entire system is designed in such a way that allows for abstracted customisation of a users scene and lighting. Moreover, the effects system is designed for extensibility. This allows users to write shaders and output their own screen-space effects.

The post-process effects that are currently part of renderer like bloom, depth of field and greyscale are an example of the kind effects available in many established packages. The Unity Game Engine, for example includes many of these techniques. Perhaps most important to this iteration was the solution implemented for anti-aliasing. It is a feature seen as an integral part of any renderer that is expected to produce a good aesthetic.

Chapter 4

Appendices

Appendix A

1. **Horizontal Blur Code**

```
uniform sampler2D u_Sampler;
uniform float u_pixel;

varying vec2 v_TextureCoord;

void main ()
{
   vec4 sum = vec4(0.0);

   // blur in y (vertical)
   // take nine samples,
   // use distance blurSize between them
   sum += texture2D(u_Sampler,
        vec2(v_TextureCoord.x - 5.0*u_pixel,
                v_TextureCoord.y)) * 0.04;
   sum += texture2D(u_Sampler,
        vec2(v_TextureCoord.x - 4.0*u_pixel,
                v_TextureCoord.y)) * 0.04;
   sum += texture2D(u_Sampler,
        vec2(v_TextureCoord.x - 3.0*u_pixel,
                v_TextureCoord.y)) * 0.08;
   sum += texture2D(u_Sampler,
        vec2(v_TextureCoord.x - 2.0*u_pixel,
                v_TextureCoord.y)) * 0.12;
   sum += texture2D(u_Sampler,
        vec2(v_TextureCoord.x - u_pixel,
                v_TextureCoord.y)) * 0.14;
   sum += texture2D(u_Sampler,
        vec2(v_TextureCoord.x,
                v_TextureCoord.y)) * 0.18;
   sum += texture2D(u_Sampler,
        vec2(v_TextureCoord.x + u_pixel,
```

```
                v_TextureCoord.y)) * 0.14;
    sum += texture2D(u_Sampler,
          vec2(v_TextureCoord.x + 2.0*u_pixel,
                v_TextureCoord.y)) * 0.12;
    sum += texture2D(u_Sampler,
          vec2(v_TextureCoord.x + 3.0*u_pixel,
                v_TextureCoord.y)) * 0.08;
    sum += texture2D(u_Sampler,
          vec2(v_TextureCoord.x + 4.0*u_pixel,
                v_TextureCoord.y)) * 0.04;
    sum += texture2D(u_Sampler,
          vec2(v_TextureCoord.x + 5.0*u_pixel,
                v_TextureCoord.y)) * 0.04;

    gl_FragColor = sum;
}
```

2. **The final blend of the Anti Alias Shader**

```
// WEIGHT
//e_kernel.x (0.5)
float finalWeighting = (1.0 - depthWeight *
    normalWeight) * u_kernel;

//SMOOTH
vec2 offset = v_TextureCoord * (1.0-finalWeighting);

vec4 s0 = vec4(texture2D(u_ColorMap,
    offset + top * finalWeighting).xyz, 1.0);
vec4 s1 = vec4(texture2D(u_ColorMap,
    offset + bottom * finalWeighting).xyz, 1.0);
vec4 s2 = vec4(texture2D(u_ColorMap,
    offset + right * finalWeighting).xyz, 1.0);
vec4 s3 = vec4(texture2D(u_ColorMap,
    offset + left * finalWeighting).xyz, 1.0);

gl_FragColor = (s0 + s1 + s2 + s3)/4.0;
```

Bibliography

[1] Smith, P (2011). Deferred Rendering.
NCCA, Bournemouth.

[2] Rost, R (2010). Opengl Shading Language. 3rd ed.
Boston: Addison Wesley. p101-461.

[3] Hargreaves, S. Harris, M. (2004). 6800 Leagues Under the Sea.
Available: http://http.download.nvidia.com/developer/presentations
/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf.
Last accessed 17th May 2011.

[4] Shishkovtsov, O (2005) Deferred Rendering in S.T.A.L.K.E.R
Available: http.developer.nvidia.com/gpugems2/gpugems2_chapter09.html
Last accessed 11th Aug 2011