Deferred Shading with Stereoscopic Rendering

By Thomas James Russell

Supervisor: Laurent Noel

A project report submitted in partial fulfillment of the degree of BSc (Hons.) Computer Games Development

21st April 2011

Abstract

Architecting a real-time graphics engine capable of supporting large quantities of dynamic lights is a difficult challenge when also taking into consideration the development of an efficient batching system. Deferred shading provides a solution to this issue, at the cost of material flexibility. Light pre pass, an extension of deferred shading, allows the lighting benefits of deferred shading to be achieved with additional material flexibility at the cost of an additional scene render.

Stereoscopic entertainment has recently increased in popularity with the release of new 3D filtering technology. With the industry now opening up to 3D games, having stereoscopic 3D a core feature of a game, can give it an advantage over the competition as consumers embrace the new technology.

The goal of this project was to develop a graphics engine capable of supporting many dynamic lights, by implementing light-pre-pass pipeline, in addition to stereoscopic rendering, to provide the user with a more immersive experience. The renderer needed to be efficient, and so optimization techniques such as culling and batching were implemented.

This report discusses the development of the project, from design to implementation, and presents the Duality Engine as a real-time graphics engine capable of the initial requirements.

Table of Contents

I Introduction

1.1 Background

Real-time rendering is a core element of game engines. To ensure an engine performs well at real-time rates, the render process must be efficient and optimized well. With the vast appeal of increasing quantities of dynamic lighting in real-time scenes, deferred shading is becoming more popular among graphics engine developers. In addition, with the recent resurgence of interest in 3D entertainment, stereoscopic rendering can provide an advantage over other games on the market.

1.2 Overview

This report highlights the development of the Duality Engine, and discusses the technology used to implement its main features.

In Chapter 2, an overview of recent advancements in real-time rendering of dynamic lighting is presented. This Chapter evaluates standard deferred shading, and discusses alternatives such as the light-pre-pass renderer. Chapter 3 elaborates on this, by presenting how light-pre-pass was implemented in the Duality Engine, and discussing any issues during development. Chapter 4 focuses on lighting and shadows, through lighting models, shadowing techniques the light types implemented in the Duality Engine. Chapter 5 overviews the theory behind stereoscopic rendering, the increasing popularity, and discusses the integration of stereoscopic rendering with the engine. The Duality Engine's core component, the renderer, is discussed in Chapter 6, from design through to implementation and evaluation. This is followed by the render process of DirectX11, in Chapter 7. Chapter 8 presents the development of the Duality Engine itself, from design to implementation, and evaluates the development experience. Chapter 9 presents the results of bench marking tests performed to analyse the performance of the Duality Engine on different hardware. Finally, Chapter 10 evaluates the development of the entire project.

II Deferred Shading Techniques

2.1 Introduction

2.1.1 Context

Rendering a scene with dynamic lighting in real-time is a core renderer feature for most modern real-time renderers. As pixel shading techniques increase in complexity, previous methods of calculating the dynamic light contribution on geometry are too slow for scenes with many lights due to pixel overdraw. In some cases, the light contribution is calculated by rendering the scene geometry in multiple passes and combining the lighting until the contribution from every light is calculated (Engel, 2009).

In standard forward shading, due to the lighting being coupled with the geometry, the average time taken to render a single frame is directly proportional to the number of lights affecting the geometry and the complexity of the geometry (Hargreaves, 2004).

For a scene with eight lights, a forward renderer would use a shader generated for that number of lights. Generating shaders for each material with each number of lights suffers from combinatorial explosion. With twelve lights and four different material types, this would combine to be forty-eight different shaders for four materials. This does not take into account different light types. This raises issues with architecting an optimized rendering system using batching, as the batch order would need to be by light quantity and type.

2.1.2 Overview

Using a deferred shading approach, a scene can be rendered in real-time with a high number of lights irrespective of scene geometry complexity (Valient, 2007). Section 2.2 presents the concept of the deferred shading approach, benefits of using deferred shading and known implementation issues. Section 3 presents three alternative solutions which develop the deferred shading concept.

2.2 Deferred Shading

2.2.1 The Deferred Shading Concept

The standard forward rendering technique to render a scene using dynamic pixel lighting involves calculating the most influential lights when rendering geometry (Placeres, 2006). A common approach for applying many lights is to apply different shaders for each material for different quantities and types of lights. Due to the tight coupling between geometry, materials and lighting calculations, scene complexity is proportional to the number of objects combined with the number of applied lights (Hargreaves, 2004).

Figure 2.1.1 An in-game night scene from GrandTheftAuto IV. Car and street lighting is calculated using a deferred approach.

An issue with performing lighting calculations when rendering geometry arises when new geometry is rendered over existing geometry. This overdraw impact can be lowered using a deferred approach.

The deferred shading technique was first proposed by Deering (1988), though the technique wasn't labelled "Deferred shading" until later (Hargreaves, 2004). By storing the data required to complete the lighting equation, the lighting can be decoupled from the rendering of geometry, this allows the lighting to be applied as a post-process (Placeres, 2006).

The deferred shading concept is composed of three main stages: the *geometry* stage – where the scene is rendered and material data is stored, the *lighting* stage- where the lighting is calculated, and the *composition* stage – where the lighting is combined with the material from the geometry stage.

As the lighting calculation is removed from the geometry stage, the implication of overdraw is minimised, as lighting calculations won't be wasted on pixels which don't appear in the final render. This allows a higher number of lighting calculations per pixel than a standard forward renderer, and thus more lights.

2.2.2 The G-Buffer

The lighting stage requires the position data and normal data for each pixel to apply the lighting as a post-process. This data is gathered into a collection of textures during the geometry phase, known as the Geometry Buffer (or "*G-Buffer*") to allow the equation to be completed at a later stage (Akenine-Möller, Haines, and Hoffman, 2008).

Frank Puig Placeres (2007) proposed the G-Buffer structure shown in Figure 2.2.2 as an initial starting point for deferred shading. However, this structure does not store any material properties, such as the diffusive colour of the object, or how reflective the material is. Furthermore, the same lighting model must be applied to each pixel during the lighting stage (Engel, 2009).

For instance, if the Blinn-Phong lighting model was used to construct the lighting accumulation, the entire scene would be lit using Blinn-Phong shading. This may not be desired, as different lighting models are suited to different materials, i.e. Minneart for a silk dress.

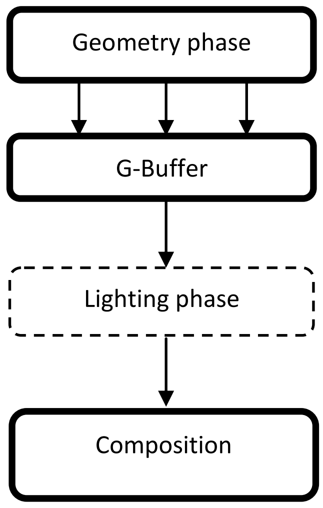


Figure 2.2.1 Deferred shading pipeline (courtesy of Frank Puig Placeres)

	R	G	B	A
Texture 1	Position X	Position Y	Position Z	[empty]
Texture 2	Normal X	Normal Y	Normal Z	[empty]

Figure 2.2.2 Example G-Buffer layout (courtesy of Frank Puig Placeres)

To solve this problem, a value could be stored in either the position or normal texture's alpha channel to indicate which lighting model to use, although these channels may be required to store material information as shown in Figure 2.2.3.

	R	G	B	A
Texture 1	Normal X	Normal Y	Normal Z	Scattering
Texture 2	Diffuse Colour R	Diffuse Colour G	Diffuse Colour B	Emissive
Texture 3	Specular Intensity	Specular Power	Occlusion	Shadow amount
Texture 4	Depth	Depth	Depth	Depth

Figure 2.2.3 Example G-Buffer layout (courtesy of Shawn Hargreaves)

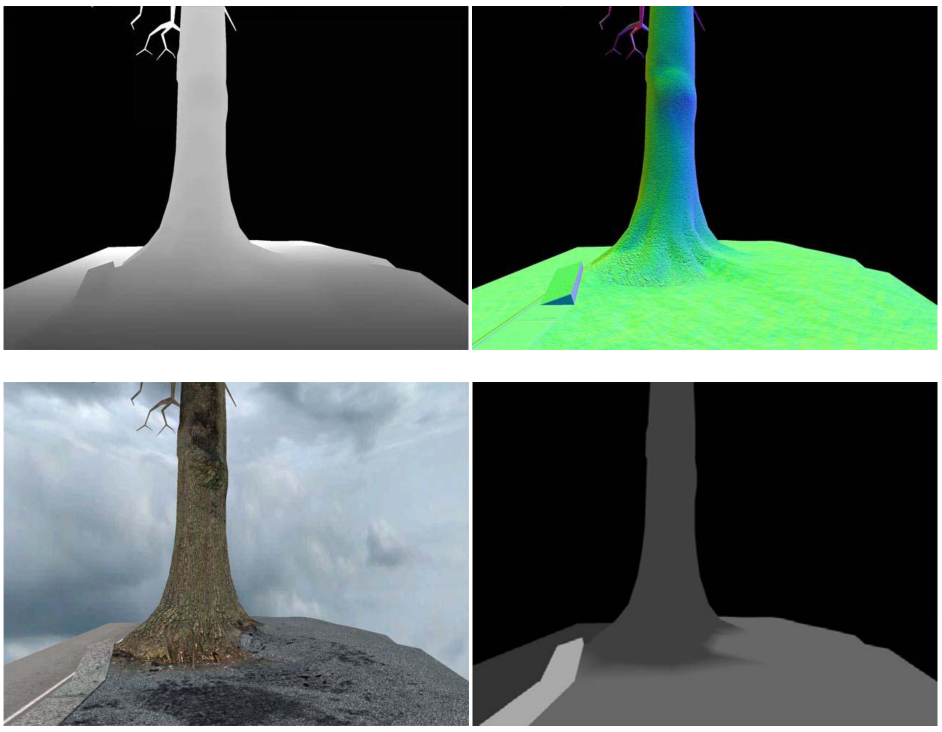


Figure 2.2.4 Example texture components of a G-Buffer; depth (upper left), normal (upper right), diffuse (lower left), specular intensity (lower right). (Courtesy of Shawn Hargreaves)

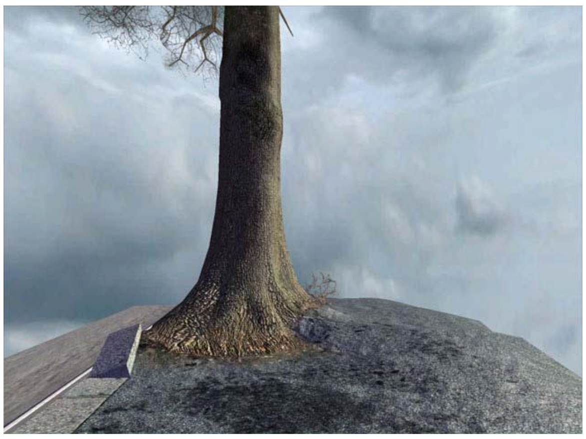


Figure 2.2.5 Example composition of lighting accumulation with G-Buffer textures. (Courtesy of Shawn Hargreaves)

To specify material attributes, the G-Buffer must contain more textures as demonstrated in Figure 2.2.3 and Figure 2.2.4 with the addition of diffusive colours, specular properties and other material attributes. These material attributes are then combined with the accumulated lighting to create the final frame image as shown in Figure 2.2.5.

The depth of each pixel can be stored rather than the position, as the viewspace and worldspace positions can be reconstructed using the depth (Hargreaves, 2004).

This allows a single value to be stored instead of three, so more material data can be stored in the texture by lowering the precision of the depth value stored. Shown in Figure 2.2.3.

2.2.3 Issues with Deferred Shading

As the lighting is decoupled from the rendering of scene geometry, all material attributes which directly affect lighting must be output to the G-Buffer (Engel, 2009). As a result the flexibility of materials is tightly dependant on the available memory of the platform.

The G-Buffer is composed of several textures which are output from the geometry stage, so the platform must support Multiple Render Targets, otherwise the scene geometry must be processed for each texture (Engel, 2009). Aside this requirement, as each texel must be written for each texture in the G-Buffer, deferred shading often suffers from a high fill rate requirement. Where a standard forward approach would merely output a single colour value, a deferred approach would output four sets of values per pixel.

Due to the G-Buffer storing the data per-pixel, only the closest fragment will be stored in the G-Buffer, which results in semi-transparent objects overriding the geometry data behind (Pangerl, 2009). For this reason semi-transparent geometry is not stored when writing the G-Buffer.

2.2.4 The Benefits of Deferred Shading

The most noticeable benefit of deferred shading is the availability of high quantities of dynamic lights within a scene. Alongside this, popular post-processing techniques have the desired texture inputs from the G-Buffer to not require rendering the scene using multiple passes. Depth of field, ambient occlusion and fog can sample the G-Buffer textures without needing to gather new data. New graphics techniques such as soft shadows calculated in screen space also benefit from using these textures (Gumbau, Chover and Sbert, 2010).

The lighting stage of deferred shading can also be interlaced with an existing forward renderer given the renderer supports a depth only pre pass. This was illustrated by Malan (2009) during the production of Crackdown, where the lighting for street lamps and car headlights were applied after the scene had been processed.

2.3 Alternate Solutions

The standard deferred shading concept stores material data at the pixel level and then calculates the accumulated lighting for each pixel using the material data. However, if the lighting can be calculated prior to rendering the geometry, the amount of data needed to bridge the lighting and geometry stage is minimised.

2.3.1 Light Indexed Deferred Rendering

Damian Trebilco (2009) proposed a solution to separate the lighting and geometry rendering stages; similar to standard deferred shading, but with light accumulation calculated prior to geometry rendering.

Trebilco modified the standard deferred shading pipeline to include a geometry depth-prepass, where the depth buffer is filled. This is then sampled, and used to reconstruct the position in view space of each pixel.

The lighting stage then iterates through the lights, each of which have a unique index, and decides if the light affects that viewspace position. If the position is affected by a light, the index of that light is stored in a Light-Index buffer.

During the geometry rendering stage, the pixel stage samples the light-index buffer to detect which lights affected the geometry, and uses the light index to look-up the light properties in light data textures. The light data textures contain data such as; the position, colour and attenuation.

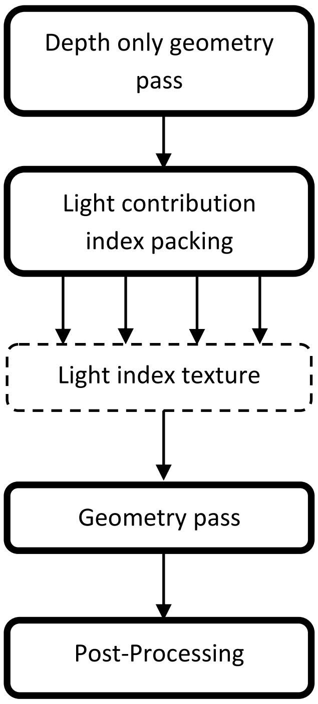


Figure 3.1.1
Example pipeline proposed by Damian Trebilco

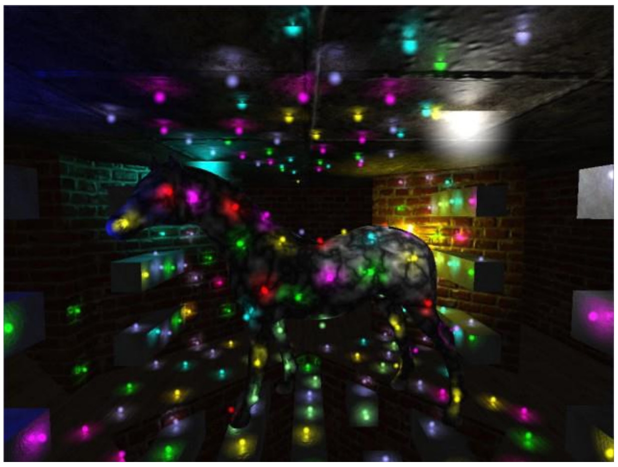


Figure 3.1.2
Example of Light Indexed Deferred Rendering
(Courtesy of Damian Trebilco)

Unfortunately, as the index is stored per pixel, the index can be overwritten by a later light which also affects the geometry.

(Trebilco 2009) derived a solution whereby the light index of four lights is packed into the texture's R,G,B and A channels, however this still means only four lights can contribute to a single pixel's lighting, unlike standard deferred shading where the number of effective lights per pixel is not bound by the storage method.

2.3.2 Light Pre-Pass Deferred Rendering

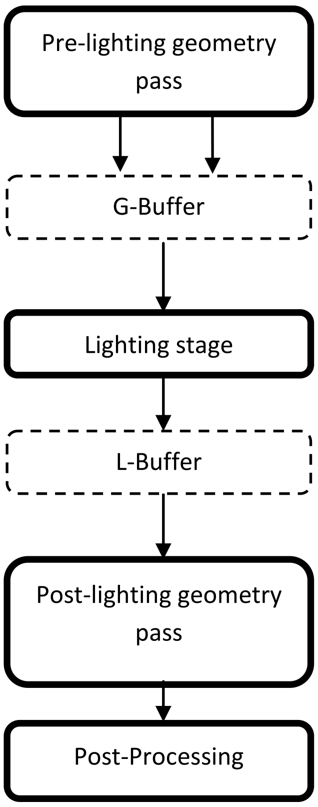


Figure 3.2.1
Example pipeline proposed by Wolfgang Engel

Wolfgang Engel (2009) introduced the concept of the light-pre-pass renderer as a solution to minimize the size of the G-Buffer. If lighting is calculated prior to geometry rendering, material data does not need to be stored, as this can be applied during the post-lighting geometry stage. This results in the light pre-pass concept having greater material flexibility than the standard deferred shading concept.

Where a standard deferred shading renderer would typically store the final light contribution for each pixel in the Light Buffer (*L-Buffer*) before combining the L-Buffer with the G-Buffer, Engel (2009) proposes storing light properties as terms of the lighting calculation. As this approach does not complete the equation, but outputs the lighting terms, the production of the L-Buffer requires less processing.

The original technique proposed by Engel (2009) only uses a single texture and does not store a specular component, as the lighting properties require four values to be stored independently. As the specular component should ideally be combined with the lighting buffer, an extra texture to store specular properties will be needed.

Applying the specular term to the fourth channel of the texture allows the diffuse and specular terms to be stored together, at the cost of the specular terms not modelling realistic specular lighting perfectly (Engel, 2009).

Figure 3.2.2 Light Pre Pass rendering used in Blur.

The light pre-pass concept allows the calculation of different lighting models to be evaluated at the post-lighting geometry stage and so allows further flexibility in material types. The concept could be adapted to store a closer approximation to the specular term, including specular light colour, however this would require six channels, three for diffusive properties and three for specular.

2.3.3 Inferred Lighting

Scott Kircher and Alan Lawrance (2009) present an extension of the Light Pre Pass concept. Utilizing the result of separating the lighting from the material and geometry stages, Kircher and Lawrance proposed that calculating lighting at a different resolution to the final render is possible with minimal visual artefacts. By calculating the lighting for a texture at 60% the size of the final rendered image, 40% of the lighting calculations need not be calculated. This optimization allows for more lights, or for the processing to be used elsewhere.

Unfortunately, during up-scaling the light texture when sampling at the final geometry stage, visual artefacts appear where the edges of rendered polygons lose definition.

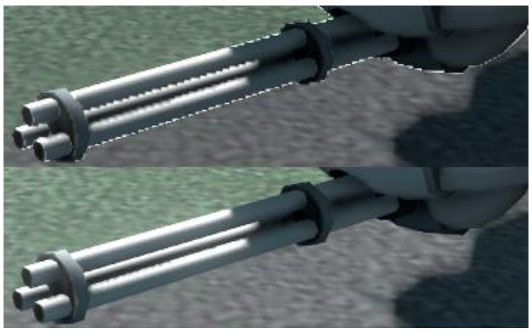


Figure 3.3.1
Example artefacts due to up-scaling the light buffer (Top).
Using the DSF to solve this issue (Bottom).
(Image courtesy of Kircher, S. and Lawrance, A.)

Scott Kircher and Alan Lawrance (2009) propose using a Discontinuity Sensitive-Filter (DSF) applied to the lower resolution image as an effective edge-detection between discontinuous polygons. This filter technique requires a unique object ID and polygon ID to be generated for the geometry and stored per vertex. The DSF decides if two pixels are edges if their IDs do not match.
When combined with the up-scaling of the L-Buffer, this solution solves many of the artefacts.

Inferred lighting has the potential to require less computation and thus perform faster than Light-Pre Pass implementations of the same resolution, as the resolution of the L-Buffer can be scaled down (Brown, 2009).

Developing the work of David Pangerl(2009), lighting for four layers of semi-transparent geometry can be calculated using a stippled pattern when writing the geometry to the G-Buffer. This pattern can then be reversed at the final geometry stage to re-produce the four layers of transparency with lighting, using the DSF (Kircher, Lawrance, 2009). This is an elegant solution to the transparency problem deferred shading renderers typically suffer from.

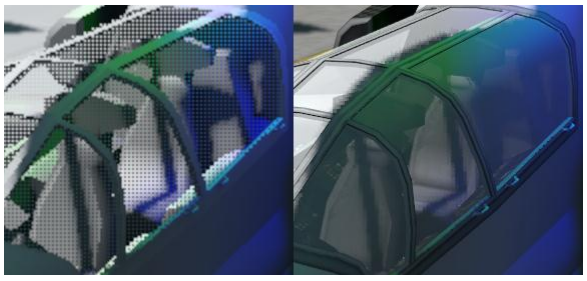


Figure 3.3.2
Example of stippled pattern when rendering semi-transparent geometry (Left), and the transparency solved using the DSF (Right).

2.4 Conclusion

Rendering a scene with many dynamic lights takes less processing time when using a modified deferred shading pipeline over standard forward rendering. Moreover, it’s possible to increase flexibility of materials – and lighting models, by deferring lighting calculations to a later geometry pass, as shown with the Light-Pre-Pass and Light Indexed designs.

Due to the limitation of a light overlap boundary, a Light-Indexed deferred shading implementation may not provide enough lighting accuracy when a scene contains many overlapping lights; yet it will perform much faster than a standard forward approach for the same scene, and require less video memory than a standard deferred shading solution.

In a contrasting situation where the accumulated lighting must be accurate, and the target platform does not support multiple render targets, a Light-Pre-Pass implementation will perform faster than a standard deferred approach, with less visual errors where lights overlap.

Finally, by lowering the resolution of the light buffer and using a smart edge-detection filter, lighting calculations can be performed on fewer pixels and thus increase performance, at slight accuracy cost. This concept could equally be applied to the standard deferred approach, along with the Light-Indexed approach.

III Deferred Shading using Light-Pre-Pass

3.1 Section Introduction

Dynamic multi-light rendering has been a major issue to consider when architecting many real-time 3D rendering systems. With the introduction of deferred shading into industry-leading game engines, deferred shading has increased in popularity as the core modern multi-light solution. This sections presents how Light Pre Pass deferred shading was implemented in the project.

3.2 The Light-Pre-Pass Pipeline

The Light-Pre-Pass renderer modifies the standard deferred shading pipeline to increase material flexibility and minimize bandwidth issues. Where the standard deferred shading pipeline requires all material data to be output at the geometry stage and stored in the geometry buffer, in the light pre pass pipeline, only the data required to complete the lighting equation is required.

At the lighting stage in the pipeline, the contribution of each individual light source is calculated and accumulated into the light buffer. It is common for the light buffer to contain both diffuse and specular contributions to the light equation.

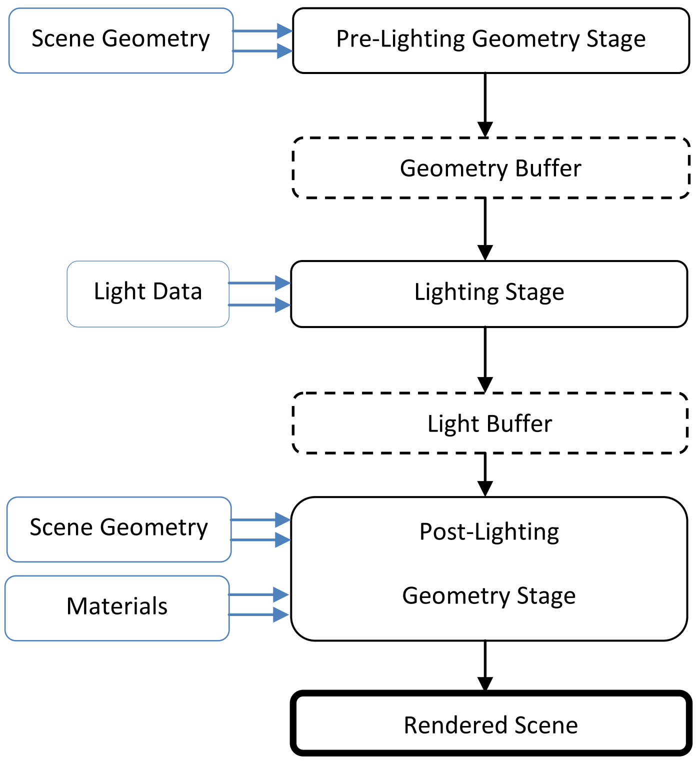


Figure 3.2.a Rendering pipeline in Duality Engine

In the final stage, which is a standard forward shading geometry pass, the final lighting values for each pixel can be sampled from the light buffer, and either contribute, or not, to the final pixel colour.

As the final geometry stage is simply a forward shading pass, additional lighting can be calculated in the standard way. This can be useful in situations where the data required for the lighting stage isn't available, and so the data in the light buffer will be incorrect. This is common when rendering semi-transparent geometry, as only one layer of data can be stored in the geometry buffer. By using this forward pass to additionally send extra light data, transparency issues with standard deferred shading implementations can be minimized with the light pre pass pipeline.

3.3 Design

Whilst the layout of the Geometry Buffer and Light Buffer aren't as essential to the light pre pass design to the standard deferred shading design, they still required consideration and planning for optimal performance.

3.3.1 The Geometry Buffer

As the necessary terms required to complete the standard lighting equation are position and normal, these were required to be stored in, or be calculated from the data in the G-Buffer. Instead of storing the position data for each pixel, the position can be recalculated by reprojecting the depth values, and transforming them from view space to world space.

As DirectX 10 through DirectX 11 allows sampling of depth stencil buffers, the depth value wasn't required to be output to an additional render target. Outputting to multiple render targets is one of the core bottlenecks in standard deferred shading implementations. This was an issue with the early DirectX9 prototype, which used a deferred shading pipeline, rather than the light-pre-pass pipeline.

The ability to read the depth stencil buffer and reconstruct the world position of a pixel using the depth, allowed the G-Buffer structure shown in figure 3.3.1.a to be used.

	R	G	B	A
Texture 1	Normal X	Normal Y	Normal Z	SpecularPower
Texture 2	Depth	Depth	Depth	Depth

Figure 3.3.1.a G-Buffer layout used in the renderer

As the Duality Engine was initially designed for low to mid-level hardware, the texture formats used in the G-Buffer were R8G8B8A8. This worked, but produced visible errors where the lighting calculations would produce block-shaped patches as shown in Fig 3.3.2.a. To remove this visual artefact, a higher precision floating point texture format was used, R16G16B16A16.

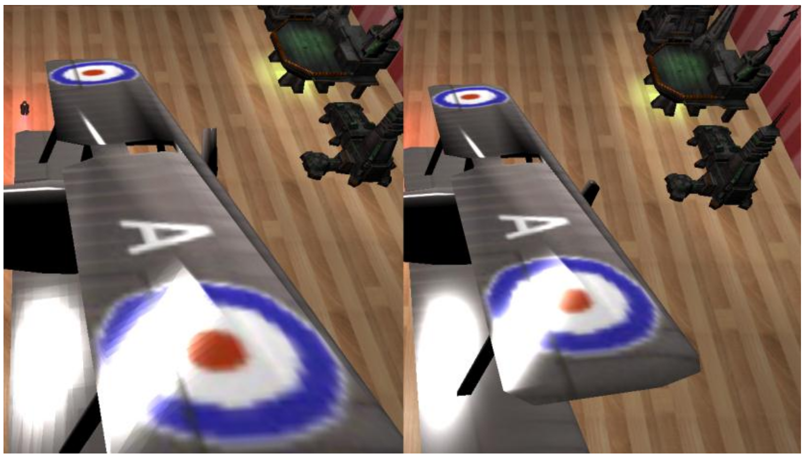


Figure 3.3.2.a

Left: Block-shaped lighting issue when storing normal in an integer format texture.

Right: Issue solved using floating point texture

Changing the texture to a higher precision format meant the float inaccuracies were minimized; however this was not optimal for low-end hardware, as higher precision texture formats may not be supported.

Additionally when storing the normal, as the floating point texture couldn't store the sign of each value, each value became unsigned, producing errors. It was required to pack the values into the range 0 to 1, rather than the unit length range of -1 to 1 for each component.

The design of the G-Buffer and the rendering pipeline differs from the designs presented by Engel (2009). Where Engel suggested deferring the completion of the lighting equation to the final stage, the post lighting geometry stage, the renderer in the Duality Engine completes the lighting equation at the lighting stage. This however, means that more data needs to be stored in the G-Buffer, in particular the specular power of the material. In addition, only one lighting model is supported at the lighting stage, the phong model discussed in chapter 4, section 2.1.

3.3.2 The Light Buffer

It was necessary to store the diffuse lighting contribution of each light, so that materials can complete the equation in the final stage. The diffuse lighting contribution was a three-float colour value, calculated for each light and accumulated for all lights. As the light buffer resource was a four-float format, an additional value could be stored in the alpha channel. However, the specular component was still required to be stored, as usually materials have independent diffuse and specular colour values.

There were two solutions to this storage problem, either an extra texture buffer could be created to store the specular contribution, or the specular contribution could be packed into the final value of the initial buffer.

	R	G	B	A
Texture 1	Diffuse (R)	Diffuse (G)	Diffuse (B)	Specular strength

Figure 3.3.2.a Specular contribution stored as single value in last channel of light buffer.

	R	G	B
Texture 1	Diffuse (R)	Diffuse (G)	Diffuse (B)
Texture 2	Specular(R)	Specular(G)	Specular(B)

Figure 3.3.2.b Specular contribution stored as full colour in additional texture.

As storing the specular in an alternate texture would require either rendering to multiple render targets or using a two-pass approach, it would not only require more memory, but it would also require more processing. However, this would allow the specular colour to be stored independent of the diffusive colour.

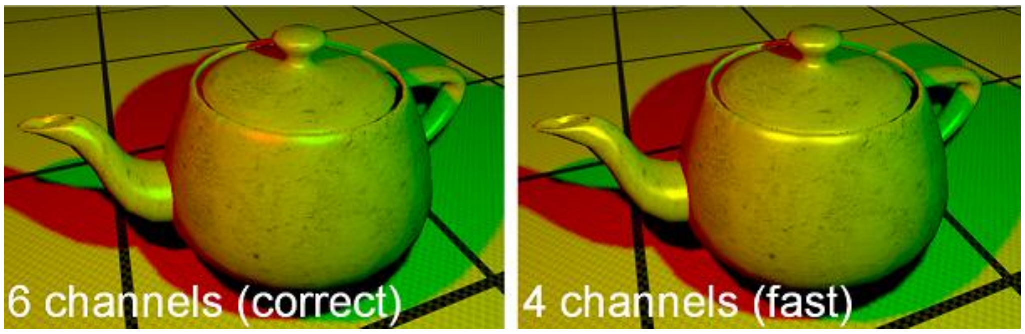


Figure 3.3.2.c Storing the specular contribution in 3 channels (left), versus storing the specular contribution in 1 channel (right). Notice the loss of colour in the red specular highlight. Image courtesy of Crytek.

It was decided, that for the purpose of this renderer, efficiency had precedence over accuracy in this situation. This allowed the light buffer to remain as a single resource, and also meant multiple render targets weren't required.

3.4 Implementation

3.4.1 Pre Lighting Pass

The pre lighting pass was implemented to fill the G-Buffer with the necessary data to complete the lighting stage. Each material which later used the deferred lighting calculated at the lighting stage was required to output the normal in world space and the depth when rendering the pre-pass. This was implemented by ensuring effects which would later sample the light buffer contained a technique labeled "pre". This technique would only need to output the world space normal, as the depth was read from the depth stencil buffer. For some materials, this was just directly writing the normal vector streamed into the pixel stage. However, for other materials which modified the normal at a pixel level, such as normal and parallax mapped materials, the modified normal would need to be written. As there was no way to ensure this from the engine's perspective, it was assumed the pixel shaders output the correct normal values for lighting.

3.4.2 Lighting Pass

During the lighting stage of the pipeline, the contribution of each light source is accumulated in the light buffer. To calculate a light source's influence on a given pixel, a pixel shader is executed. For every pixel to be processed, geometry must be rendered which encapsulates these pixels. This task could be performed on the CPU by modifying individual pixels in the light buffer; however this task is more suited to the GPU.

To complete the lighting equation, data is needed about the surface the light comes in contact with. As described in section 3.3.1, this data is stored in the G-Buffer which can now be accessed at the lighting stage. By sampling the G-Buffer at the location of the pixel currently being processed, the data required for that pixel can be accessed, and the equation completed.

As the position in world-space of each pixel was not stored in G-Buffer, the position was reconstructed using the depth buffer, and the inverse of the camera's view projection matrices. This process is described in chapter 7, section 2.3. Once the world space position is calculated, the lighting equation can finally be complete, as the world-space normal is stored in the G-Buffer.

During the lighting stage, only one lighting model is used; Blinn-Phong, and the lighting equation is completed at this stage. The original design for light-pre-pass by Engel (2009), calculated the sum of common terms, which could then be used in the forward pass (see section 3.4.3), to complete the equation. Whilst Engel's design allows the use of a combination of different lighting equations during the forward pass to complete the lighting, the lighting stage in the renderer limits this to Blinn-Phong and only stores the result of this calculation. This results in the lighting stage being slightly slower than usual, but with the forward geometry pass being a lot faster. Finally, to accumulate the result of the lighting equation, additive blending is used when rendering the geometry which encompasses the pixels.

3.4.3 Forward Pass

After the lighting pass, the scene is rendered again, using a traditional render of the materials. As the lighting has already been calculated, the shaders can simply sample the light buffer at the same pixel location to acquire the accumulated lighting for that pixel. By rendering the scene again, new data can be gathered in the usual methods for the materials without extending the G-Buffer. This allows the light pre pass renderer to be more flexible when with respect to materials. Additionally, as the post-lighting geometry pass is similar to the traditional render pipeline, more data can be passed to the material. This data could be light data which is used by the material in interesting lighting models not used at the lighting stage. In example, the Fresnel term discussed in chapter 4, section 2.2.

3.5 Conclusion

Implementing the light-pre-pass pipeline provided rendering capabilities of large numbers of dynamic lights in a scene, by separating the lighting from the scene geometry. Where traditional deferred shading stores the material data in the G-Buffer, the light-pre-pass pipeline used an additional rendering pass to gather new data to calculate the final colour of each pixel.

Only the specular strength was stored in the light buffer, rather than the specular colour values. This optimisation allowed for faster light accumulation, as the lighting shaders did not need to output to two buffers. However, this optimization does not reflect reality entirely, and this leads to unrealistic lighting where the specular colour and diffuse lighting colour would be different. In addition, instead of storing the lighting terms in the light buffer, the completed result was stored. This meant the lighting model used to complete the lighting equation needed to be defined at the lighting stage. As different lighting models can vary on the number of input variables, this implementation of different lighting models would soon suffer from the same issues as deferred shading, where the G-Buffer is increased in size to accommodate for more data. To improve on this, the terms for the equation could be stored, as shown by Engel(2009), with the equation resolved in the forward pass. This would allow greater material flexibility in terms of the lighting model used.

As the implementation of the light pre pass pipeline in the Duality Engine doesn't use multiple render targets, the renderer should be able to support low-end hardware and certain integrated chips, without intense modification of the implementation. This means, that with this implementation a larger audience can use this software. This would be advantageous in a commercial situation.

IV Lighting and Shadows

4.1 Section Introduction

Lighting plays an important role in graphics applications; it can draw the viewer's focus onto important objects in the scene, and enhance realism. Different lighting models can be used in different situations, with some lighting models providing fast approximations over physical correctness. Dynamic shadows also enhance the realism of a scene, and provide the viewer with visual information about the proximity of objects.

4.2 Overview of Lighting Models

4.2.1 Blinn-Phong Lighting

The Blinn-Phong lighting model was used as the primary shading model during the lighting stage described in chapter 3, section 4.2. The Blinn-Phong shading model is composed of three terms; ambient, diffuse and specular.

$$L_a = a_{colour} \times m_{colour}$$

where:
L_a is the ambient term
a_{colour} is ambient lighting colour
m_{colour} is material colour

The ambient term is a constant value which approximates the average brightness of the scene, and the light reflected from other objects. The ambient term is usually constant throughout the scene, however some implementations define materials having an ambient colour (Schüler, M., 2009).

$$L_d = (n \cdot l_{dir}) \times l_{colour} \times m_{colour}$$

where:
L_d is the diffuse term
n is the world-space normal
l_{dir} is the direction to the light
l_{colour} is the light's colour
m_{colour} is material colour

The diffuse term is an approximation of the reflected light being scattered. The diffuse term is usually accompanied by a colour representing the absorption of light waves, or efficiency of reflection.

$$L_s = (n \cdot h)^{m_i} \times l_{colour} \times m_{colour}$$

where:
L_s is the specular term
n is the world-space normal
h is the half-way vector between the direction to the light and the direction to the camera
l_{colour} is the light's colour
m_{colour} is material colour
m_i is the material's specular intensity

Finally the specular term also represents the reflectance of light waves, but with much less scattering than the diffuse term. The specular intensity can vary in reality between lights, in addition to materials, however, in this implementation only the material specular intensity was considered. Blinn modified the phong equation by using a halfway vector as shown on the left. This modification closer reflects reality (Schüler, M., 2009).

The lighting terms are brought together to complete the equation:

$$L = (a_{colour} \times m_{colour}) + ((n \cdot l_{dir}) \times l_{colour} \times m_{colour}) + ((n \cdot h)^{m_i} \times l_{colour} \times m_{colour})$$

or

$$L = l_a + l_d + l_s$$

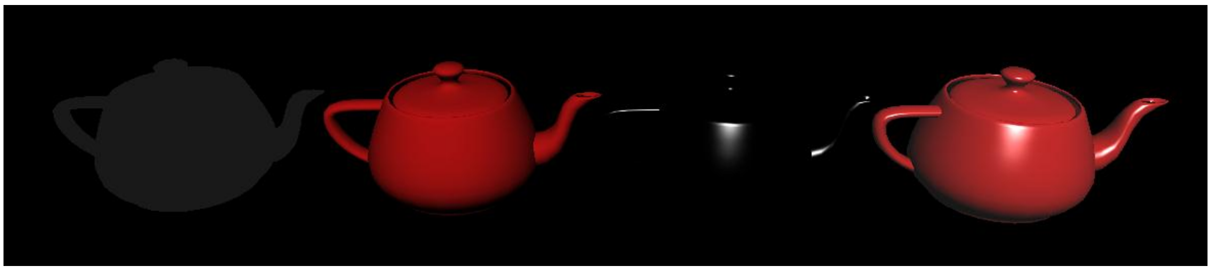


Figure 4.2.1.a:
Blinn-Phong in Duality Engine: ambient, diffuse, specular, complete Blinn-Phong, respectively.

4.2.2 Fresnel Lighting

Fresnel lighting can be used to simulate the sub-surface reflectance of light. By comparing the viewing angle to the world-space normal, a value is created can be used to reflect the subsurface scattering of materials. This value is often combined with the reflection of the environment and can be used to simulate water (Akenine-Möller, Haines, and Hoffman, 2008).

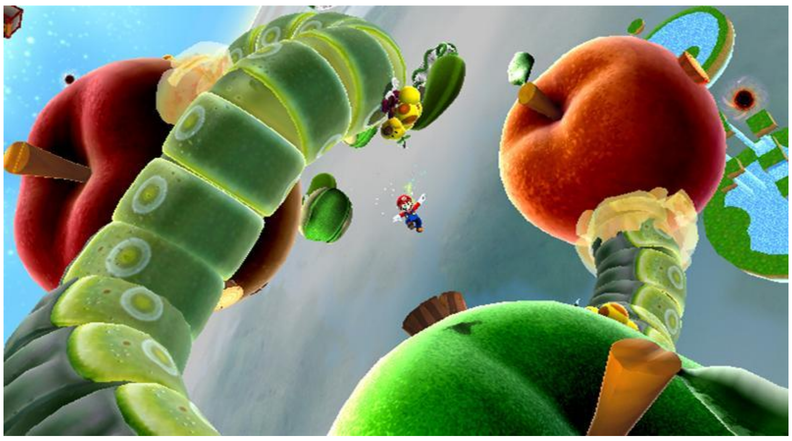


Figure 4.2.2.a – Fresnel lighting used in Super Mario Galaxy.

However, replacing the environment with solid colour leads to a rim lighting effect which can be seen in figure 4.2.2.a and figure 4.2.2.b. This concept has been applied in a large number of recent games, from the Super Mario Galaxy series, to Team Fortress 2 and the Left4Dead series.

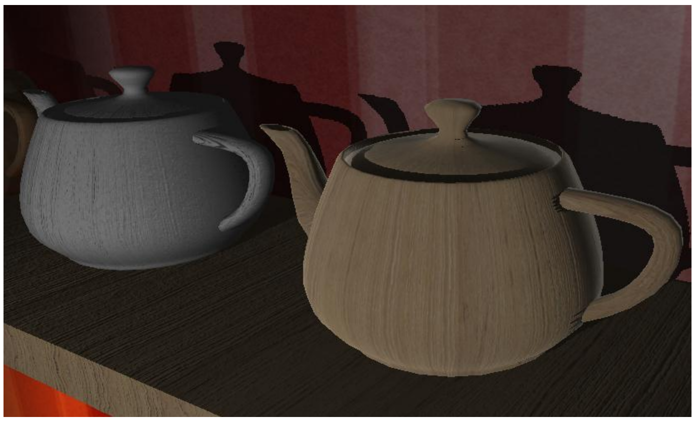


Figure 4.2.2.b Fresnel lighting used in Duality Engine.
Right teapot with Fresnel term, compared with left teapot without Fresnel term.

4.3 Directional Lights

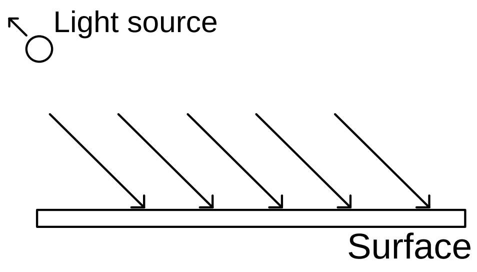


Figure 4.3.a Illustration of directional light sources. The light source is infinitely far away, making the light vector parallel for all surfaces.

Directional lights can be considered as being a light source of infinite distance to a surface, (Rabin, S., 2008). As the light source is infinitely far away, the vector of separation is parallel for all surfaces. This results in light affecting the entire scene equally, disregarding shadows.

When integrating directional lights with deferred shading, it was necessary to consider how the light contribution could be calculated for every pixel. The solution to this was to generate a quad which covered the entire viewport. This ensured that every pixel was processed when the quad was rendered. A quad was rendered for each directional light in the scene, accumulating the resulting light influence. This process is described in more detail in Chapter 3 sub-section 3.4.2.

4.4 Point Lights

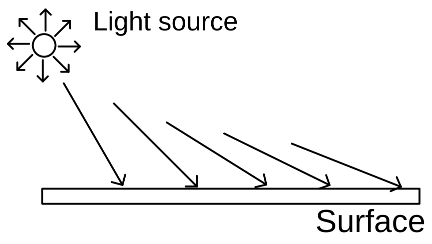


Figure 4.4.a Illustration of point light sources. The light source has a finite position and emits light in all directions. The light vector is not parallel for all surfaces.

Unlike directional lights discussed in section 4.3, point lights have a finite position defined. As a result the vector of separation between the light source and surfaces is not the same, thus not parallel for every surface. As point lights are not of infinite distance, attenuation of the light strength is taken into consideration with the surface's distance.

When integrating point lights with the deferred shading renderer, initially the contribution of each point light was processed for every on screen pixel – similar to directional lights. This was wasteful, as in most cases, a single point light would not contribute greatly to every pixel on screen. As the point light equation takes into consideration attenuation, if a point light source is far enough from the pixel's world location, the actual contribution of that light will be unnoticeable. Similarly, if a point light source has a low brightness level then the light will contribute to distant pixels a lot less.

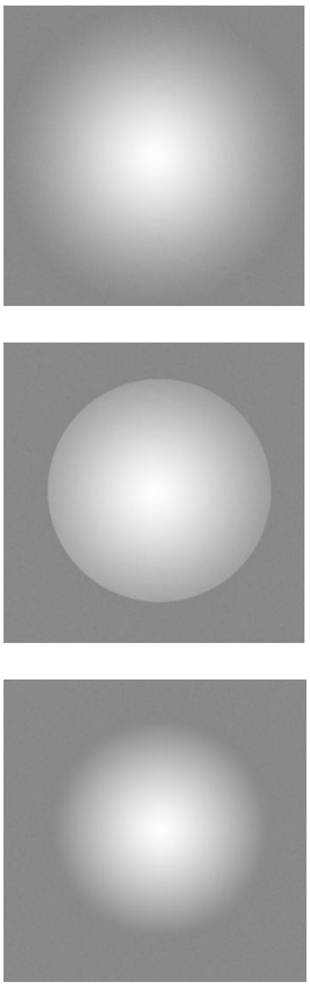


Figure 4.4.b.
Top, calculating lighting with fullscreen quad.

Middle, calculating lighting with sphere mesh.

Bottom, modified attenuation with sphere mesh.

To solve this issue, instead of rendering a quad which filled the viewport for each point light, a sphere mesh was rendered at the location of the light. To accommodate for different levels of brightness in the lights, the sphere was scaled proportional to the brightness level. This introduced a number of visible artefacts where the lighting would naturally extend beyond the sphere's radius, and create harsh edges in the silhouette of the sphere geometry. As shown in figure 4.4.b.

This issue was solved by defining a maximum distance at which the point light's influence would falloff to zero. This solution redefined the attenuation, and modified the lighting equation. While this solved the issue, the attenuation of the lights became unrealistic. For the Duality Engine however, this was acceptable, as efficient rendering is more important than ultra-realism.

Rendering spheres in place of point lights also raised another visual artefact when the camera entered inside the light's influence radius. The lighting would suddenly disappear as the geometry for the sphere was clipped against the near-plane of the frustum. It is very common in real-time graphics applications and games that the camera enters a light's influence radius, and so this issue needed a solution.

To solve this issue, front face culling was enabled when rendering the sphere geometry. This ensured that only the rear facing geometry would be rendered, i.e. the back surface of the sphere. From the camera's perspective, this encompasses the entire volume of the light's influence radius, from all visible angles, and allows the camera to enter the radius without the light disappearing.

Unfortunately, as the front faces are not being rendered, but the rear faces are, this lead to errors where the light's geometry would intersect standard scene geometry. This was due to depth checks the against the scene's depth buffer.

Whilst using the depth buffer to cull the light geometry and reduce lighting calculations was important, it was more important that lighting was constant throughout the scene. For this reason, depth testing was disabled when rendering point lights.

4.5 Spot Lights

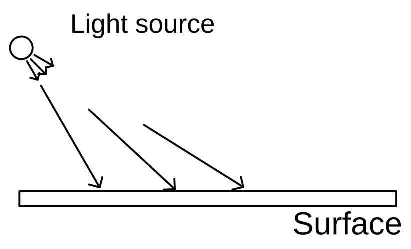


Figure 4.5.a Illustration of directional light sources. The light source has a finite position and emits light within a cone in a given direction. The light vector is not parallel for all surfaces.

Similar to point lights discussed in section 4.4, spot lights have a finite position resulting in non-parallel light vectors for surfaces. Unlike point lights, spotlights only affect a cone region of the scene's surfaces. This cone region is usually determined by a cone angle, the smaller the cone angle, the thinner the cone.

As the cone angle was made dynamic for flexibility, this meant it would be difficult to simply place a cone mesh in place of spot lights as the geometry would need to be generated for each spot light which encompasses the lights influence. Because of this, spot lights are applied by rendering quads which fill the viewport. This issue could be solved, by generating the geometry at run-time at the geometry stage, before the pixel processing stage where the spot lights contribution is calculated. This optimization would decouple the processing of spot lights from the resolution of the output.

Soft spotlights were calculated by extending the spot light equation to include an inner radius, the point at which the falloff would begin. The falloff was calculated by linearly interpolating between the inner radius and cone angle vector, by the distance to the light's facing vector of the pixel.

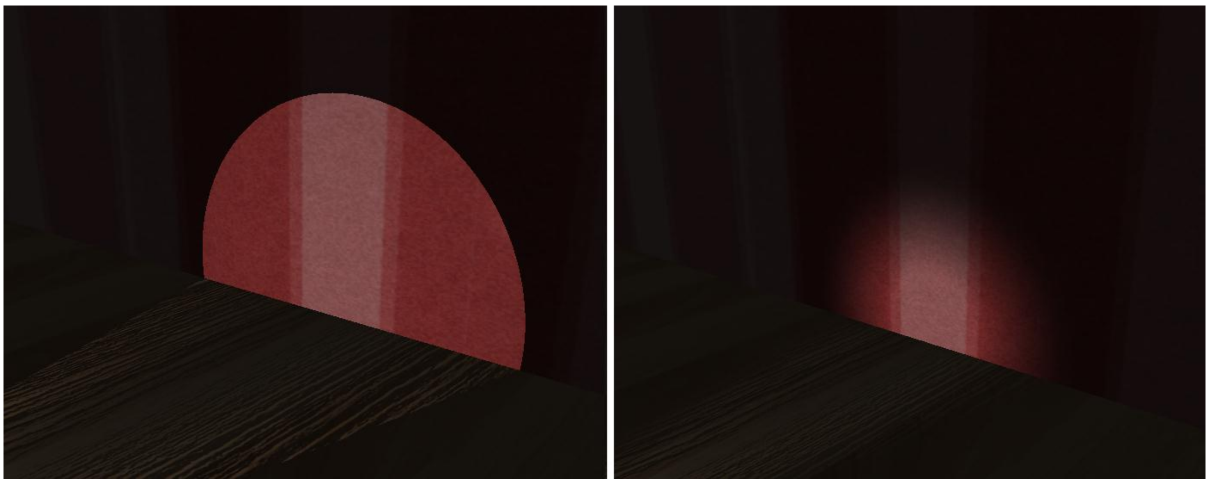


Figure 4.5.b. Soft edge spotlights in Duality Engine. Left: Spotlight with hard edge. Right: Spotlight with soft edge by interpolating between inner radius and outer radius.

4.6 Shadow Mapping

4.6.1 Process Overview

Shadows allow the viewer to perceive the spatial structure of a scene more accurately than a scene without shadows. By using shadow techniques, a scene can appear more realistic, and increase viewer immersion. One particular technique, known as shadow mapping was used in the Duality Engine.

Shadow mapping, sometimes known as shadow texturing, is the process of creating a depth texture, and performing depth checks against this texture when applying lighting (Akenine-Möller, Haines, and Hoffman, 2008). To find if a surface is occluded from a light source, a depth comparison with other scene geometry is required. The scene is rendered from the perspective of the light source, storing only the depth in a depth texture. A depth check is performed against the depth texture, using the distance of the current pixel to the light source. If the distance is greater than the depth stored, then the pixel is behind the occluder with respect to the light source's perspective, and does not receive light.

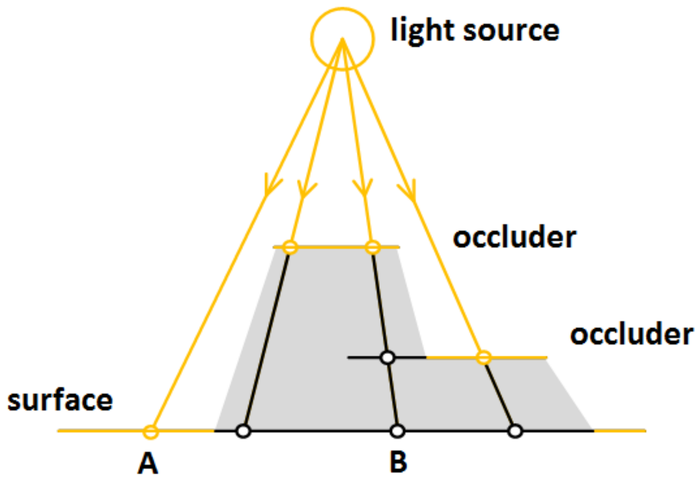


Figure 4.6.1.a Illustration of the shadow mapping method, using depth checks to detect if a surface is hidden by an occluder

Comparing the depth values of contact points on a surface with the depth values stored in the shadow map, determines if a contact point receives light from the light source. In this example shown in figure 4.6.1.a, contact point A has the lowest depth value to the light source, as there is no occluding geometry. However, for contact point B, the occluding geometry is preventing the surface from receiving light.

4.6.2 Implementation

The entire scene is rendered from the light's perspective. For this, the scene would need to be transformed into the shadowed light's local space, rather than world space. A unique combined view projection matrix was needed for each shadowed light source. These matrices were used in exactly the same process as with the camera's combined view projection matrices discussed in chapter 7, section 2.2.

Only depth is needed to perform the depth comparison, and so in an initial implementation, the scene was simply rendered to a depth stencil surface, and not to a render target. This produced some interesting incompatibility issues with DirectX11. It was discovered in DirectX11 that, when the primary render target is not set, the depth stencil buffer set must match the primary depth stencil surface size, otherwise the viewport is offset. However, by enabling the primary render target, the depth stencil buffer set is not offset, and produces expected values.

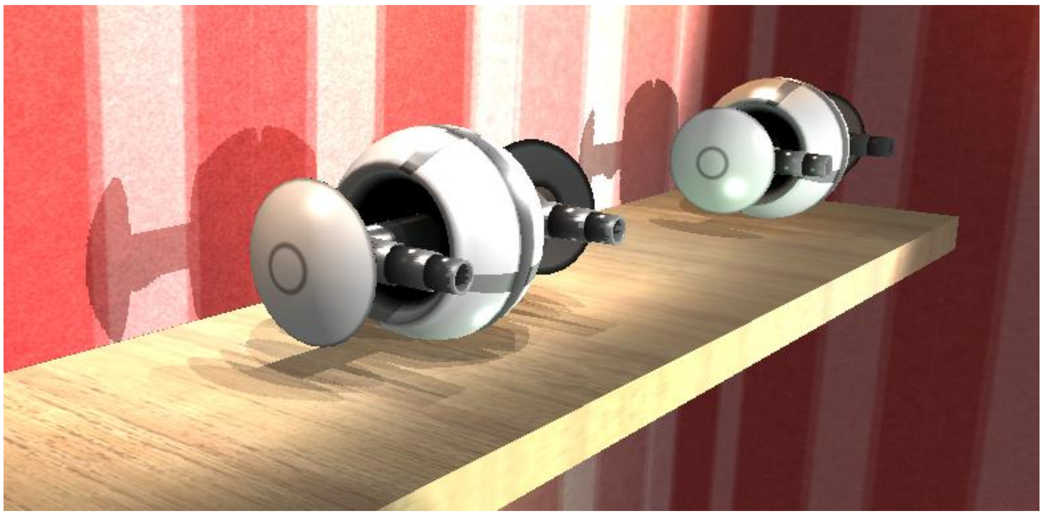


Figure 4.6.2.a Shadow mapping in Duality Engine.
This image shows the toys casting many shadows from multiple light sources.

At the lighting stage, when the light contribution for the shadowed light source is calculated, the distance of each pixel to the light source must be compared with the depths in the depth stencil buffer. For this, the position of each pixel must be transformed into the projected space of the light, to create a UV co-ordinate, and depth value. The depth stencil resource is sampled, using this UV co-ordinate, and finally the depths compared. With a pixel's depth being higher than the depth value in the resource at the same location, the pixel is further away and occluded by other geometry. In the implementation, if a pixel is occluded, lighting calculations stop, and the lighting process for that pixel ends. This produces notable sharp edges along a surface between occluded and non-occluded pixels.

4.7 Conclusion

Different lighting models can provide varying levels of realism, and provide unique art styles to enhance a scene. Real-time lighting can be calculated with approximations to physical interaction of light and the environment, such as the Blinn-Phong lighting model. Lighting models can also be adjusted to create interesting non-realistic effects, such as with the Fresnel implementation.

Although directional, point and spot lights were implemented; area lights are common among offline visualization and animation software, and with the separation of lighting from geometry as discussed in chapter 3- section 2, may be possible to implement without great modifications to the existing architecture.

Shadows add more realism to a scene, making the scene layout more understandable to the viewer. By rendering a shadow map for each shadow casting light source and performing depth checks when calculating lighting, hard-edged shadows can be implemented. With the current implementation, a render target is set and rendered to, to ensure the depth stencil buffer values are correct. This is inefficient, and wasteful, as the data in the render target will never be used. Calculating only the depth values, and not writing to a render target is an obvious optimization.

Currently, the entire scene is rendered for each shadow casting light. This is inefficient as geometry which may never be influenced by the shadow casting light is still rendered. The shadow map creation could be improved by using a spatial partitioning system to determine which geometry to render, or manually specifying occluding geometry for individual lights.

Finally, this shadow implementation could be further improved with the addition of soft shadowing techniques. To further enhance the realism of the scene, penumbra correct shadowing could be used, however this can be expensive (Gumbau, J. Chover M. and Sbert, M., 2010). Also, with the current implementation, only spot lights cast shadows, this could be extended to allow point lights to cast shadows, and using advanced shadowing techniques for directional lights.

V Stereoscopic Rendering

5.1 Section Introduction

Traditional rendering only provides the user with one image of information per frame. By providing two images, the user can perceive the depth of the world in greater clarity, by understanding the parallax shift of an object presented in each image. This section provides an overview of the theory behind stereoscopic rendering and the implementation, along with filtering methods to ensure each eye receives only the image created specifically for it.

5.2 Theory

An issue with standard real-time visual applications is how immersive the experience is. For some situations, the level of immersion can be greatly enhanced by providing more information about the scene to the user, such as the depth of objects in the scene. While the distance of objects could be gathered from the resulting size of an object, the user won't always know the scale of an object, for example, a small scaled object may appear the same size as a large scale object much further away. By providing unique views onto the scene for each eye, and ensuring each of these images is received by the correct eye, we can provide this extra depth information to the viewer.

In reality, when a person wishes to focus on an object, their eyes converge onto a single point. In example, if a person were to hold an object about 2 feet away in front of the head, the eyes would converge inwards to focus on this object. If the person wished to look further into the distance at the landscape, the eyes would converge less, as the focus point is much further away. As the human brain interprets the level of convergence, the focal distance and the parallax of each object in the two images, the depth of the environment is understood.

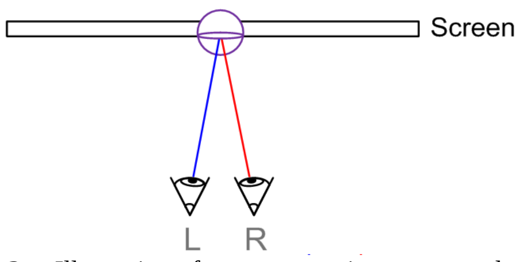


Figure 5.2.a. Illustration of eyes converging at screen depth.

As shown in figure 5.2.a, when a person focuses on an object, the eyes converge onto the surface distance to gather as much information as possible. Note the left eye converging to the right, and the right eye converging to the left. When viewing a normal flat image, the eyes will converge to focus on the screen at screen depth.

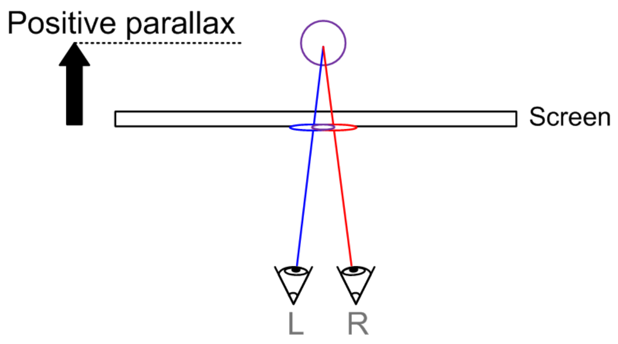


Figure 5.2.b. Illustration of eyes converging beyond screen depth.

When viewing distant objects the eyes converge to a lesser extent. This can be seen in figure 5.2.b with the eyes converging less than in figure 5.2.a. By suggesting to each eye to converge less, we can create the illusion of depth past the screen distance. This can be achieved by offsetting the image for the left and right eyes. By shifting the left eye's image to the left, and the right eye's image to the right, we can suggest to the brain to converging at this distance. This shifting is known as positive parallax, which simulates depth.

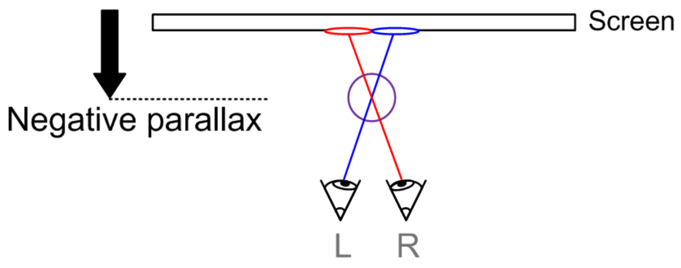


Figure 5.2.c. Illustration of eyes converging in front of screen depth.

When viewing closer objects however, the eyes converge much more, as shown in figure 5.2.c. We can suggest to the brain to converge the eyes in this manor by applying negative parallax to the two images. Negative parallax shifts the right image to the left, and the left image to the right. By using negative parallax, objects can appear to be closer than the viewing screen.

5.3 Implementation

The initial implementation of stereoscopic rendering in the project, involved simply creating two view matrices and separating them on the camera's local x-axis. As shown below:

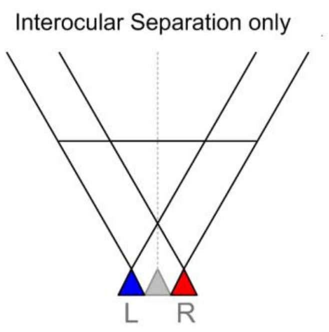


Figure 5.3.a Diagram showing the separation of view matrices.

Separating the two matrices as shown, simulates the interocular separation of our eyes in reality. These matrices are then each used to render the scene. This provides two unique views onto the scene intended for each eye. The separation of the view matrices created negative parallax for all visible objects. This occurred because as the left view position moved to the left; the result appeared that the scene shifts to the right. This was similar for the right view position, and can be seen in figure 5.3.b. This negative parallax for all scene objects meant that the entire scene would appear to be in front of the screen.

The level of negative parallax for each object in the scene was directly proportional to the depth of the object. Objects closer to the camera would produce a greater negative parallax than distant objects. With every object in the scene having only negative parallax, this became a very tiring experience. With the separation of the view matrices approaching the average interaxial of human-eyes at 6.25cm, this became intolerable.

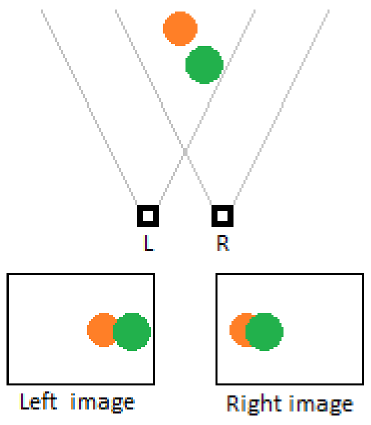


Figure 5.3.b. Diagram showing the resulting images of interocular separation

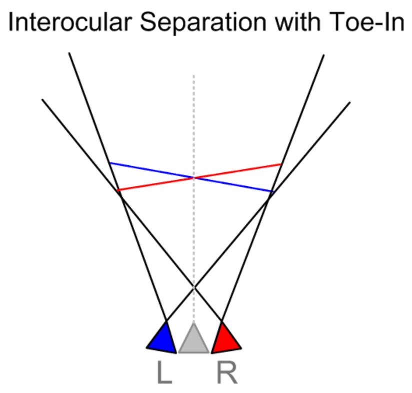


Figure 5.3.c. Diagram showing the toe-in method

To ensure that distant objects have positive parallax, the view matrices were rotated inwards on the local y-axis of the view matrix. As shown in figure 5.3.c. This process, known as toeing, creates the illusion of distant objects being far away, as the positive parallax is increased. Unfortunately, as this parallax plane is not parallel to the screen, unnatural distortions appear and this method becomes tiring for close objects.

It was necessary to ensure that any parallax shifts to persuade convergence would need to be irrespective of depth, to account for the entire scene. By applying a positive parallax shift to the entire scene for each image, the apparent depth could be pushed back into the distance, ideally behind the screen's distance.

By using a user defined convergence value and shifting the entire scene in accordance to this value, this positive parallax shift was achieved. By shifting the objects in the left image to the left, and the objects in the right image to the right by half the convergence value, the parallax value was increased.

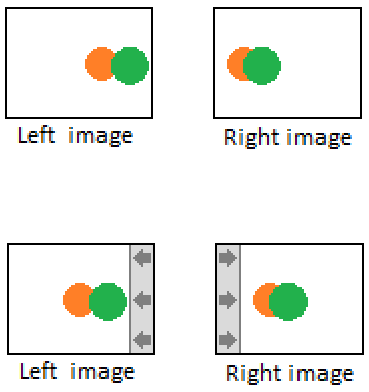


Figure 5.3.d. Diagram showing how the convergence shift is applied

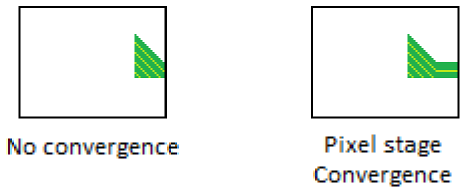


Figure 5.3.e. Diagram showing the convergence shift applied at the pixel stage, resulting in artefacts.

The convergence shift was initially implemented at the pixel stage where the two images were being combined into one display-presentable image. However, this produced banding at the sides of the image, where there wasn't enough visible data in the image to shift. This can be seen in the lower image of figure 5.3.d. As this was not a desired effect, the convergence transformation would have to be performed during the actual rendering of the geometry prior to the combination of the two images. By applying a translation to the x-component of the view space position of each vertex at the vertex stage, geometry would be translated by the convergence value. This allowed the geometry which would previously be clipped to be included in the image. This method however, required every vertex in the scene to be transformed in addition to any transforms by the standard world, view and projection matrices. It also required every vertex shader to include code to include the convergence calculation, which would be a minor irritation for the shader developers.

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Where:

n and f are near and far distances
t and b are top and bottom distances
l and r are left and right distances

Instead, the convergence shift could be applied to the projection matrix, which in turn would transform all geometry in a scene. As the projection matrix was already being combined with the scene geometry, this would incur no extra cost when rendering geometry.

In actuality, the top and bottom terms can be ignored as only horizontal parallax is needed, and the convergence value set by the user can be used here:

$$\begin{bmatrix} 2n/a & 0 & c/a & 0 \\ 0 & 2n & 0 & 0 \\ 0 & 0 & \dfrac{n+f}{n-f} & \dfrac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Where:
c is the convergence value
a is the aspect ratio

This oblique perspective matrix can be used to translate the transformed vertices along the local axis, particularly in the x and y axis. However, as we can ignore the vertical parallax, we only need translate along the x axis.

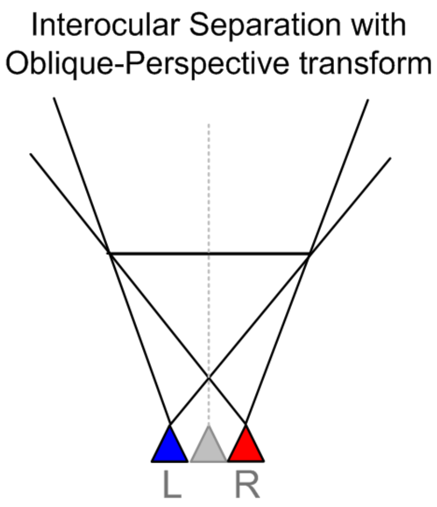


Figure 5.3.f. Diagram showing the oblique perspective transform applied to the view matrices. The frustum for each view is translated horizontally to apply the convergence shift.

5.4 Filtering

In addition to creating both the left and right images, the two images much reach the left and right eyes of the user respectively, with minimal crosstalk. Crosstalk can be reduced by using filtering methods which ensure the eyes only receive the images for the eye they were created for.

Anaglyph filtering separates the two images, by first assigning a particular light frequency for the left and right image, and then ensuring the user has lenses which filter only the specified light frequency. As red materials don't absorb red light, red light which passes through the material is more pronounced than other frequencies. This is similar for cyan. By ensuring the user has the red lens over one eye and the cyan lens over the other, it is possible to present two separate images to the eyes independently.

A major issue with stereoscopic render but more precisely anaglyph filtering is retinal rivalry. Retinal rivalry occurs when the images presented to each eye are too dissimilar, and the brain attempts to calculate which image is real. This is likely to occur in simple implementations of anaglyph filtering as the left eye receives only one colour channel, and the right eye receives an entirely different colour channel. Even with this level of retinal rivalry, simple implementations of the anaglyph filter can lead to further problems. If simply gathering the red component of the left image for the left eye, and only gathering the cyan component for the right eye, objects of those specific colours may be lost in one of the two images. This would mean that one image would contain the object, and the other, would not – leading to retinal rivalry. This can be seen in appendix E.

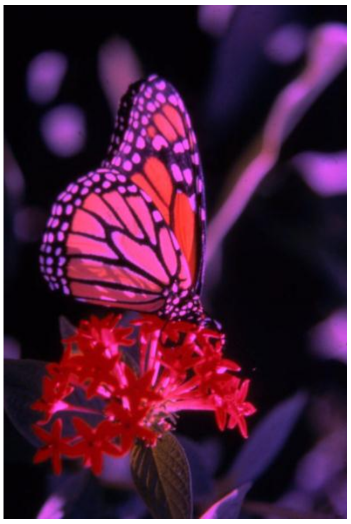


Figure 5.4.a Combined anaglyph image using simple channel gather technique. *Reference image provided under the creative commons license.*

To combat this; when applying the filter, the images where saturated to greyscale before packing into the specific colour filters. This process is shown in appendix F. The resulting combined anaglyph image can be seen in figure 5.4.b. When viewed through anaglyph glasses, both the red filter and cyan filter receive the same brightness. This reduces retinal rivalry, and provides a more comfortable user experience for viewing 3D. Unfortunately, this means a loss of all colour information, which may be unacceptable for certain situations such as colour based puzzle games or games which use colour to highlight important interactive elements i.e. Mirror’s Edge.

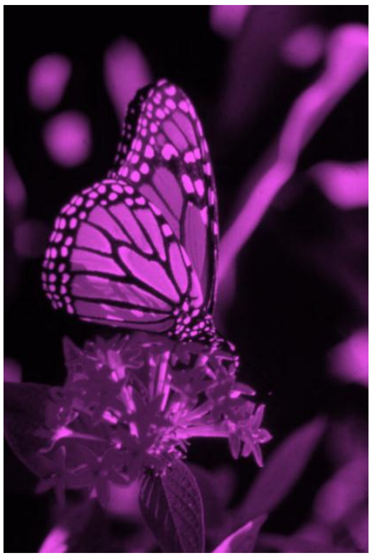


Figure 5.4.b Combined anaglyph image using colour saturation prior to channel gather technique. *Reference image provided under the creative commons license.*

Retinal rivalry can be minimised by using a filtering method which allows both eyes to receive the images in full colour. An alternative technology to anaglyph filtering which allows this is the NVidia 3D Vision kit. The NVidia 3D Vision kit uses shutter-lenses to alternate between which eye can see the screen at any one point in time. However, this technology requires a display capable of supporting 120 Hz. The NVidia 3D Vision kit was the initial target stereo hardware for the project, however the full stereo API for the hardware was not accessible, and this had to be replaced with anaglyph. Fortunately, this situation was accounted for early on at the planning stage and had minimal impact.

5.5 Conclusion

With the recent popularity increase with 3D entertainment, having an engine which can render real-time stereoscopic scenes efficiently can give a game a significant advantage. Stereoscopic rendering was a core feature of the Duality Engine. By rendering two images of the scene instead of one, the scene can be presented in stereo. To ensure the viewer received the correct image in each eye, anaglyph filtering was used to separate the image. It should be noted, that any view independent processes, such as the rendering of shadow maps (see chapter 4, section 6) , and GUI rendering (see chapter 6, section 8) was only rendered once per frame.

The implementation of stereoscopic rendering could be improved by optimizing the render process, using methods such as rendering scene objects with minimal parallax shift once, or processing similar stages of the light-pre-pass pipeline at the same time for each image. In addition, known techniques such as screen-space reprojection could be used to remove the second render pass entirely.

The Duality Engine while supporting 3D, only truly supports anaglyph filtering. This technology is considered outdated by most consumers, and can be uncomfortable for some viewers. However, the glasses are cheap to produce, and the process doesn’t require additional display hardware. This project could be extended to explore non-anaglyph filtering techniques, and experiment with shutter glasses technology. Shutter glasses would provide a richer user experience as full colour would be available, with minimal retinal rivalry.

VI The Renderer

6.1 Section Introduction

Real-time graphics applications such as games, have at the core a renderer. This renderer is the main system designed to produce an image visualizing the simulated world. In real-time environments the implementation of the renderer is vital to the performance of an engine. To ensure the performance of the engine is as fluid as possible, the renderer must be well optimised and perform efficient rendering. This section presents the design and implementation of the renderer in the Duality Engine.

6.2 Requirements

A renderer capable of rendering 3D scenes in real-time was required to complete the project. The renderer needed to support the programmable pipeline, as the fixed pipeline does not support most of the techniques which were to be implemented in the project. The renderer also needed to be efficient, as stereoscopic rendering requires two images to be generated per frame instead of one. The renderer needed to support loading and rendering meshes contained within mesh files. In addition, the scene and assets used within needed to be dynamically loaded, as the scene may change after compilation. The final major requirement was material flexibility; this was an issue with the prototype as the materials were only confined to a select few.

6.3 Design

The renderer was designed mainly around efficiency, and batching. As certain state changes can be expensive and cause bottle-necks; minimizing state-changes was considered throughout the design of the renderer. In example, by ensuring all the instances using a particular shader are rendered whilst that shader is set, the shader states do not need to be set as frequently.

A batching order was decided which ordered the state changes from most to least expensive. The initial batch order is by effect. As effects contain numerous shaders, passes, techniques and other state changes within, these are expensive to set recurrently. In a naïve implementation where batching is not used and instances are simply rendered in the order of creation, the effect passes and all states contained would be set for every rendered instance. By sorting the instances by effect, this can be minimized to only set the effect passes and states once per effect. This can lead to a significant increase in speed as redundant state changes are minimised.

The next batching order was batching by texture and material attributes. As a material can be described as a set of shaders, textures and attributes such as colour, a material stored the textures and attributes, while being linked with an effect which stored the shaders and states. By linking the materials to the effects, the effects could process the materials at render time, rendering any instances that used the materials.

Next, an instance type was designed to link the materials to the geometry. The instance type would be created by the material, and linked to a sub-mesh. Thus, the next batching order was by sub-mesh. This would allow setting the necessary vertex and index buffers before rendering the sub-mesh. Finally, any instances which used this combination of material and sub-mesh would be placed into the instance type, with only the matrix of the instance to be updated between render calls.

As the lighting is decoupled from the rendering of geometry (see chapter 3, section 2), light quantities and types need not be considered when designing the batch order, and so near-perfect batching can be achieved. This solution provides material flexibility, and also space for improvements, such as instancing, as the only data sent between render calls is the matrix transform of the instance.

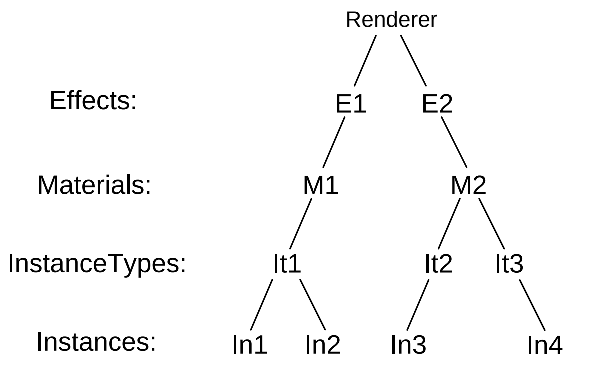
With the use of materials, textures, meshes and effects, an individual resource manager was designed to manage each of these resource types, from creation to destruction. The resource managers were designed to enable searching through the resources to find particular resources of given names and incrementally iterate through each resource also. This was designed such that when linking resources together, for example linking a material with a particular effect, the task could be simplified to the developer using the engine.

6.4 Implementation

The renderer was implemented using DirectX11. This had unforeseen issues during development which aren't present with developing applications with DirectX10. Firstly, there is no D3DXMesh interface with D3D11. While mesh support would have been directly beneficial, this meant that the creation of and loading of vertex and index buffer data had to be done manually. Instead of wasting a long time developing mesh file parsers, the Assimp importer API was used. Additionally, as the Assimp importer can import different format mesh files such as .3ds, or .obj, this became a very beneficial choice.

Another issue with D3D11 is sprite and font rendering have been removed in place of the new Direct2D API. Unfortunately, interoperability with this API from D3D11 is still inefficient, as surfaces are copied between the GPU and CPU. As the renderer was required to render dynamic text efficiently, a font rendering solution was required. Section 6.8 elaborates on the implementation of font rendering.

The process of rendering the scene can be abstracted to form a tree structure illustrated in figure 6.4a, similar to a scene graph (VanVerth, J.M. Bishop, L.M., 2004). The tree helps to visualise the process, using the batch order discussed in the design in section 6.3. Using the structure of figure 6.4.a as an example, if instance "*In1*" was visible, the effect "*E1*" would be set, followed by the material "*M1*", followed by the instance type "*It1*" being set. The instance "*In1*" would then be rendered. If instance "*In2*" is also visible, as *"E1"*, *"M1"* and *"It1"* have already been set; the instance can simply be rendered without any additional state changes.



Process Order:

1. Set effect E1.
2. Set material M1.
3. Set instance type It1.
4. Render instance In1.
5. Render instance In2.
6. Set effect E2.
7. Set material M2.
8. Set instance type It2.
9. Render instance In3.
10. Set instance type It3.
11. Render instance In4.

Figure 6.4.a Abstracted **t**ree representation of render process

This tree structure can be derived from the implementation as; each effect holds a list of materials, each material holds a list of instance types, and each instance type holds a list of instances. To improve efficiency, and remove redundant state changes, branches of the tree were clipped if they didn't result in an instance being rendered. This is further described in section 6.7.2.

6.5 Scene Management

The scene manager contained the scene hierarchy, and managed the creation of scene node objects, such as instances, cameras and lights. Methods were provided to update the hierarchy, whereby the transformations from local space to world space were calculated.

6.5.1 Scene Hierarchy

The scene was stored using a hierarchy, in addition to a linear vector of nodes. This allowed iteration of the scene nodes, but also allowed the hierarchical transforms to be maintained. In example, the turret head of a tank should stay attached to the top of the tank, when the lower half of the tank moves, rather than become separated. By grouping scene nodes by a common parent, they can be grouped together and transformed collectively. This is a useful feature during scene organisation.

By updating the base node with the world transform matrix, an identity matrix, the entire hierarchy is updated. With every node updating its own world transform, and then invoking the matrix update method of each of its children, the entire hierarchy is gradually updated and transformed into the world-space relative to its parent.

The hierarchy was also used for culling during early development, as entire collections of nodes could be ignored against frustum checks if the parent failed to pass. This was optional for each scene node, as a scene node could either check its child nodes when culled, or ignore them. Frustum culling is further described in section 6.7.1.

While the scene hierarchy is comparable to a scene graph only because of its hierarchical structure, it should be noted that the hierarchy can be composed of more than scene nodes. As cameras, lights and instances each have a spatial presence within the scene; they inherit from the scene node base type and as such provide the same node hierarchy features as the scene node. This allows them to be integrated into the hierarchy with ease.

6.6 Resource Management

Resources such as textures or shader files may be required by many entities in a game. If entities are allowed to load resources themselves, a single resource can be loaded multiple times, which is wasteful. If all entities requiring the same resource were to use only a single instance of that resource, be it a geometry file or other resource, this would reduce memory usage and increase caching. Resource management can be described as the creation and deallocation of resources and improving resource re-use. By allowing entities to request a resource from a resource manager, that resource can be shared between entities.

In the Duality Engine there are four specific resource types, textures, materials, meshes, and effects. Each of these resource types has their own resource managers. Careful consideration was required when designing the managers, as they needed to create and load resources into memory, and allow searching through the loaded resources to find if a resource was available. In each of the managers, resources are stored linearly in an STL vector. While this allows fast traversal through the resources when required, this does result in delays when extending the vector size, however this delay is usually unnoticeable with low numbers of resources. Instead of using the linear storage structure, a dictionary structure could have been used. However, to meet the low resource quantities of the requirements, this was not necessary, and would provide minimal performance increase.

It was considered to provide an interface for loading resources in a separate thread, this would allow the engine to update the game and entities whilst loading a resources. This would also allow for shorter loading screens as the resources could be loaded faster in a multithreaded environment. This was not implemented however, primarily due to the difficulty of the task and available development time. Instead, resources are loaded synchronously, with callbacks to the engine to update the loading text when loading.

6.7 Optimisation

6.7.1 Frustum Culling

Rendering objects which do not appear in the final frame is wasteful and highly inefficient. One simple method for detecting if an object is visible is frustum culling. The view frustum represents the visible portion of the world, and can be described using 6 planes. By checking an object against the view frustum, we can determine if the object won't contribute to the final image. If an object definitely won't contribute to the scene, it can be culled, and not rendered. These optimizations allow the processing of this object to be redirected elsewhere.

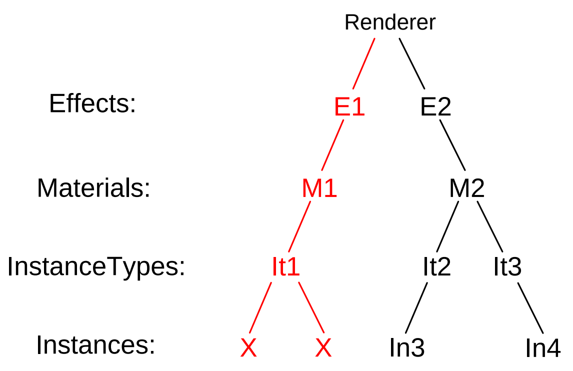
Frustum culling was used in the Duality Engine, to improve performance and minimize wasteful render calls. By transforming the world space position of each instance into the projected view space of the view frustum, the resulting position can be simply checked against all planes by comparing the linear position values to the limits of -1 to 1. Using only this position however, meant that large instances such as a skybox, or large building would be culled if the origin was outside the frustum. This was incorrect, as the scale of the instance should be considered. To account for scale when checking the instances against the frustum, the radius of the sub-mesh used by the instance was gathered. This radius was calculated by finding the furthest vertex from the local origin, and using the distance of this vertex to the centre as the radius.

In a scene with many thousands of instances, iterating through each instance and detecting if it is visible is inefficient. To optimize this, a flag can be set for all scene node types, which will cull all of the node's children should that node be culled itself. This was implemented to allow for hierarchical spatial containers. In example, if a room contains 1000 teapots, and the room is not visible, then the 1000 teapots are also not visible.

6.7.2 Minimizing state changes

The render process can be abstracted to form a tree structure as discussed in section 6.4.
When instances have been culled, and are no longer visible, state changes are minimised by removing branches from the tree which ultimately don't lead to an instance. Removing these branches means that effects, materials, or instance types won't need to be set, resulting in minimizing redundant state changes.

To determine if an effect, material or instance type needs to be set, the tree is traversed. Firstly, each effect attempts to optimise each material, should each material be optimised out or the material list is empty, the effect won't be set. During the material's optimisation process, the material attempts to optimise each instance type. If each instance type used by that material is optimised out or the instance type list is empty, that material won't be used. During the instance type's optimisation process, the instance type checks if the instances have been flagged as visible or invisible. If all instances are invisible, or the instance list is empty, the instance type won't be set and is flagged as being optimised out.



Optimised render process:

1. Set effect E2.
2. Set material M2.
3. Set instance type It2.
4. Render instance In3.
5. Set instance type It3.
6. Render instance In4.

Figure 6.7.2.a Illustration of removing redundant state changes

As presented in figure 6.7.2.a, if the instances *In1* and *In2* are not visible, the instance type *It1* does not need to be set. As *It1* is the only instance type in material *M1*, *M1* is not required to be set, and finally, as *M1* is the only material in effect *E1* and doesn't need to be set, *E1* isn't required to be set.

This process of removing the branches in the tree which don't result in an instance being rendered saves a lot of redundant state changes. The result is only necessary state changes being performed between render calls.

6.8 Font Rendering

Font rendering capabilities were removed from DirectX11 in place of the Direct2D API. The most viable solution regarding implementation time and efficiency was to develop a text rendering system.

To provide the functionality of rendering text, a bitmap image containing a character map was loaded into memory. This character map was a 16 by 16 grid of each ASCII character in succession in a particular font typeface. By understanding the layout of the characters on the character map, a method was created to calculate the UV coordinates for a given character, which was then used by the text streams when rendering the text.

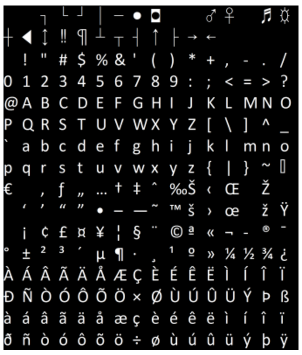


Figure 6.8.a An example character map

The text streams store an array of 256 characters. This length is long enough for most cases, with a fixed length for efficiency. A dynamic length buffer would require constant creation/deallocation of DirectX11 buffers, this seemed unnecessary as multiple buffers could be used instead.

To render the text, a buffer is created which contains the location of each character along with width and height, and the UV coordinates. Colour is also stored. The buffer data is streamed into the geometry stage, where the geometry shader creates a quad for each character, with correct UV mapping. The pixel stage then samples the character map using these coordinates, and combines the result with colour. This final colour is then blended with the final scene's render.

As each character may have different widths, using the same separation width for each character looks wrong, with large spaces between characters. To solve this issue, additional width data was loaded in and stored alongside the character map with the character data in the font type. This additional width information was used to correctly size each quad and offset each character's placement.

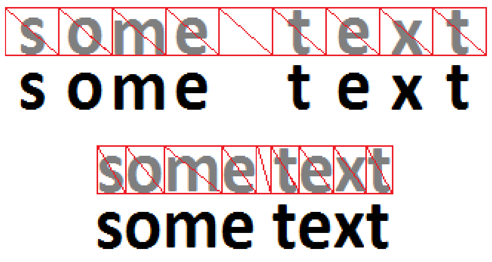


Figure 6.8.b Uniform width applied (top), individual widths applied (bottom).

6.9 Conclusion

This chapter presented how the core system of the Duality Engine, the renderer, was developed. As the renderer needed to render real-time dynamic scenes, it needed to be efficient at rendering. This main requirement was met by carefully designing an efficient batching system. This batching system allowed minimal state changes to occur between rendering instances, thus improving performance over a non-batched system. To further optimize the renderer, frustum culling was implemented for all scene nodes. This minimized redundant render calls where instances were not visible. As the culling affected all scene nodes and the renderer used the light-pre-pass pipeline (see chapter 3, section 2), lights which were not visible within the frustum were also culled. This allowed redundant lighting calculations to be minimised.

As instances could be marked not visible if culled, redundant state changes throughout the system were minimised if an instance was not visible. This was implemented by only applying state changes which would directly affect visible instances only. As this optimization was considered when designing the framework, implementation was simplified.

Using DirectX11 was an interesting learning experience. As the API did not the support loading of meshes and mesh data, this needed to be implemented externally of D3D11. In addition, as D3D11 does not currently support font rendering directly, a font rendering system was designed and implemented. In a commercial environment, the benefits of using DirectX11 over DirectX10 for this project may not exceed the difficulties of development. However, as this has now been implemented, this renderer can be reused for future projects.

Should there be additional time and resources on developing this renderer, efficiency could be improved greatly with the use of further culling techniques and partitioning schemes. Visual realism could also be increased with the introduction of advanced shadowing techniques, and advanced post processing techniques.

VII The Render Process

7.1 The DirectX 11 Rendering Pipeline

7.1.1 Input Assembler

At the Input Assembler stage of the pipeline, the data read from the input buffers, specifically the vertex and index buffers, is assembled and generated into primitives types used in later stages of the pipeline. The primitives generated can be point lists, line lists or strips, or triangle lists or strips, with additional adjacency information if required at the geometry stage. These primitives are then streamed into the vertex stage of the pipeline.

7.1.2 Vertex Stage

The Vertex stage of the pipeline reads a stream of primitive data, processes a single vertex, and outputs this processed vertex data to the next stage in the pipeline. Since DirectX10.0, the vertex stage is entirely programmable using shaders, and no fixed pipeline process exists. One common use for the vertex stage is processing data which would be too costly to process at the pixel stage, which can then be acceptably interpolated. This can be illustrated with the use of vertex lighting. Vertex shaders can also be used to transform geometry in regular or irregular fashions, such as with animation and bone weighting, where the position of individual vertices is influenced by a collection of bone parameters. In most cases, vertex shaders are used to transform each vertex from local space to world space, and then from world space to projected view space.

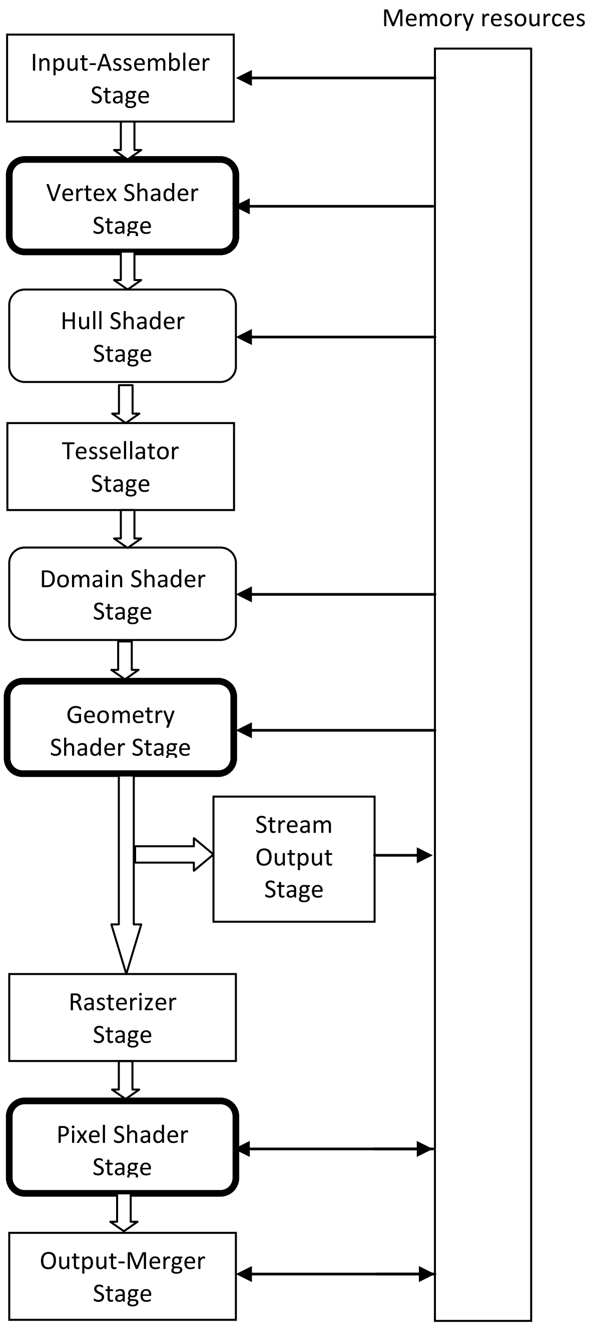


Figure 7.1.a The DirectX 11 rendering pipeline.

7.1.2 Hull, Tessellator and Domain Stages

In DirectX11, additional pipeline stages have been added to increase the flexibility of the actions an application can make the GPU perform. These stages; the Hull Shader stage, the

Tessellator stage, and Domain Shader stage can be used for generating additional geometry using a process labeled tessellation. Some uses for tessellation, are to increase the geometric detail of a mesh using a displacement map. The displacement map is usually a two dimensional array with data comprising a height scalar and displacement vector. At the end of the domain shader stage, the new vertex data is streamed into the next pipeline stage, the geometry stage. These three additional stages however, are optional, with standard implementations which can be conceptualized as a miniature fixed pipeline.

7.1.3 Geometry Stage

The Geometry stage of the pipeline, takes as input entire primitives, optionally with additional adjacency information. The geometry stage can be programmed using geometry shaders. Using shaders, the geometry stage can be programmed to generate vertices, and form primitives. An example of this is presented in chapter 6, section 9, with the generation of quads for each character in a string when rendering text.

7.1.4 Pixel Stage

The Pixel stage then processes each individual pixel composing the primitives. At this stage, the output from the vertex stage or geometry stage is interpolated for each resultant pixel. The pixel stage can be programmed using pixel shaders. These shaders must output a value to write to a render target, however that value may be zero. Pixel shaders are used to perform per pixel operations which would be to inaccurate at the vertex stage. In example, the Blinn-Phong lighting model implementation presented in chapter 4, section 2.1, is calculated at the pixel stage.

7.2 Overview of 3D Transformations

When a mesh is created, each polygon is defined by a set of vertices. These vertices represent a point in space relative to the origin of the mesh. However, in games and dynamic 3D graphics applications, it is necessary, to have the mesh drawn in different places, and with different orientations than was defined when the mesh was created.

7.2.1 Local to World Space

The standard approach to placing and rotating a mesh within a virtual world is to transform each vertex position from local space into world space. By transforming each vertex position individually, the entire mesh is transformed as a whole. This can be achieved by combining the vertex position with a world matrix storing the orientation and position relative to the origin of the world.

After this transformation, a mesh can be rendered in world space relative to the world origin. However, as the view into a scene is usually not from the origin of the world, with the view moving dynamically through the world, additional steps are required for a true dynamic 3D graphics application.

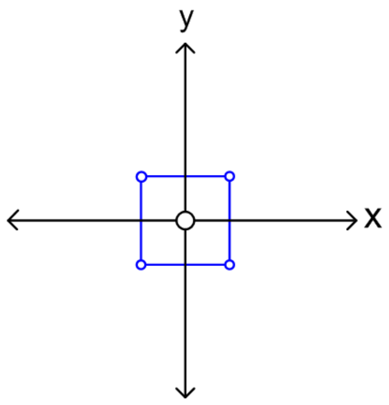


Figure 7.2.1.a Cube mesh in local space

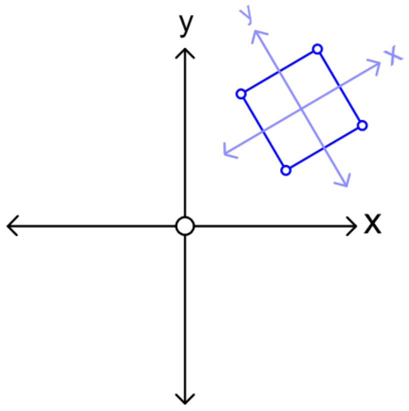


Figure 7.2.1.b Cube mesh in world space

7.2.2 World to View Space

To create the effect of a camera moving through the world, the meshes must be rendered relative to the camera. To do this, a view matrix must be constructed which is then combined with the world matrices of every mesh instance. The view matrix is the inverse of the camera's world matrix.

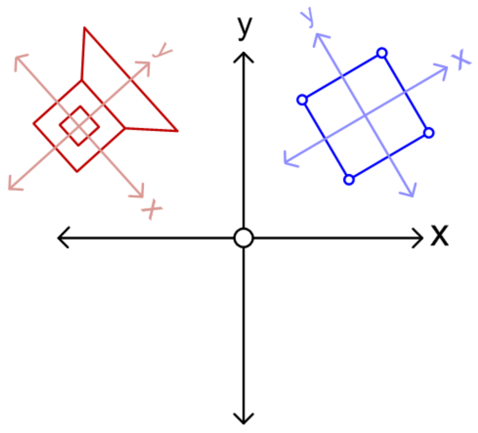


Figure 7.2.2.a Cube mesh in world space. Camera (red) in world space.

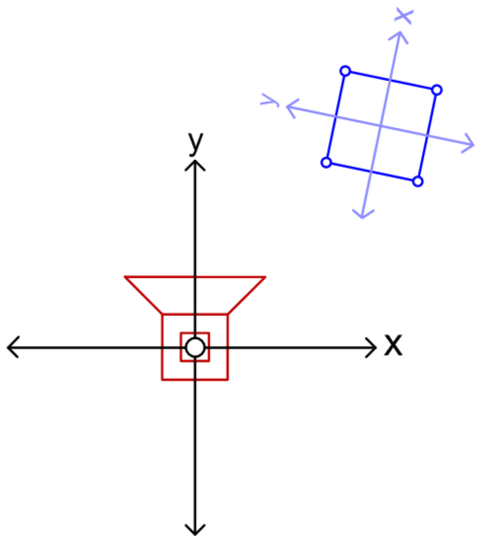


Figure 7.2.2.b Cube mesh in camera space. Camera (red) in local space.

7.2.3 Reconstructing World Space Position using Depth

When rendering the scene, a depth buffer element is filled for each pixel. When depth testing is enabled, the value in the depth buffer is compared with the value for each pixel. If the depth test passes, the new depth overwrites the stored depth. A common use for this is to determine if a polygon is behind another polygon, so a resolution can take place to ensure the closest polygon is drawn. Each of the values stored in the depth buffer corresponds to a pixel location, which was initially determined by transforming a world space position into a projected view space position using a combined view projection matrix. As the screen-space position can be transformed by the inverse of the combined view-projection matrix, the original world space position can be recalculated.

The screen-space position can be described as four-component vector, with the x and y co-ordinate ranging from -1 to 1, and the depth as the z value. As the original w value was not stored in the depth buffer, it must be recalculated. This can be calculated at the vertex stage, with the results interpolated for the pixel stage when reconstructing the depth. After the screen-space position is constructed, it must be transformed by the inverse view-projection matrix. This reconstructs the original world space position.

VIII The Duality Engine

8.1 Section Introduction

Often, after a successful game has been developed, published, and distributed, a studio may wish to create a sequel to the game. If the developers can reuse code used in the original game, they can distribute their time working on new features, and enhancing the game. By separating the core functionality of the code from the game specific code, the ability to reuse the code is increased. A game engine implements the functionality of the application which isn’t specific to one game. In example, the code to render a scene can be decoupled from the gameplay code. By reusing this non-game specific code, development time can be used elsewhere in future projects.

8.2 Requirements

The Duality Engine needed to be an API capable of creating a dynamic real-time interactive scene. The engine was required to support large quantities of dynamic lights. It was also required to render two images of the scene for stereoscopic rendering.

It should be noted that the engine needed to be able to render more than one scene, as it will be reused in later projects. To render unique scenes after compilation, the scene data needed to be loaded in dynamically, along with any resources used to render the scene.

8.3 Design

The Duality Engine was primarily designed for programmers with some existing experience with graphics engines. Due to it being an engine, reusability played a key role in the design and implementation, with the purpose of the engine not being to create a single demo, but to be used for future projects also.

Instead of designing the core engine class to contain all of the functionality of the engine, modules were created to focus on core aspects of the engine. In example, the entity manager was designed to create, update and remove entities from the scene. The window manager was designed to handle the initialization and update of the application window, including processing and forwarding windows messages to other modules. The input manager was designed to read user input from the keyboard, mouse and an optional four Games for Windows game controllers. The logger class was designed using the singleton design pattern, with the intent of allow log messages to be created and stored if an error occurs within the engine.

As the engine was designed not just for the scope of this project, but for other programmers to use, the interface was kept simple, providing access to the underlying managers, and common procedures such as initialization, shutdown and updating the engine.

8.4 Implementation

Following the design of the engine, and learning from past experiences of developing the prototype, the Duality Engine was developed. The implementation accurately reflected the core design of the engine, with additions made for flexibility of materials and additional features, such as shadowing.

The Duality Engine is composed of many different APIs and libraries. As discussed in section 8.7, TinyXML was used for XML parsing. As discussed in chapter 6 section 4, Assimp importer was used to read the geometry information from files. The effect11 framework was also used primarily as an automated shader system, and the math library created by Laurent Noel was also integrated.

To maintain visual feedback during long loading times, a simple callback function was implemented taking by parameter, a string containing a message to display. This combined with the start and finish loading methods provided useful information on the progression of loading.

8.5 Entity Management

Whilst using scene nodes to describe the position, orientation and scaling of a renderable instance; behavior cannot be captured with just this representation alone. By having an object which controls these scene nodes, complex behavior can be simulated, such as driving a car, or traversing a mountain. By applying behavior to these entities, the entities can then manage themselves with respect to positioning and controlling scene nodes, alongside interacting with the world. In the Duality Engine, the entity manager stores a collection of entity references. Entities can be created by the manager, with a unique ID generated for each entity. Upon creation of an entity, the unique ID is returned rather than a pointer to the entity object itself. This is to promote the use of the IDs over direct pointer access, which can lead to runtime errors if the pointer's address value is incorrect.

8.5.1 Entity Update

Instead of updating the entity data in the main game loop, outside of the engine, the entity manager calls an update method owned by the entity base class. Passing through the time step for the engine, this allows each entity to be updated by the actual time, and should remove irregular speed issues on different machines where the processing speed is different.

As each entity is individually updated, the complex behavior of entities can be managed by the entity itself, rather than relying on external method calls. This promotes better encapsulation and to a degree, code reusability.

8.6 The Logger

During development it was required to create a logging system to log errors and unusual program behavior. The logger was particularly useful in release executable builds, where debug information was inaccessible. By logging error messages, such as a missing resource files, the errors could be solved efficiently without wasting time deciphering the root of the problem.

The logger was implemented following the singleton design pattern (Rabin, S., 2008.) and was made available to all classes provided they include the logger header file. The logger can redirect log messages to the output window of the IDE, or to an event file. Outputting error messages to the event file was extremely useful during the testing stage and throughout development.

8.7 XML driven setup

Using a data driven approach to scene setup and resource loading, increased the productivity during development, as the data can be read and resources loaded dynamically. This meant that should the scene change, the entire application didn’t need to be recompiled. This allowed scene to be altered externally at runtime. Using data driven setup however, meant that there is additional processing when parsing the documents and loading the resources, but the increase in the development speed negated this.

When selecting the format to store the data, it was necessary to consider how the data would be changed. Initially, without the use of tools, the data would be changed manually in a text editor. This meant the format would need to be human readable. XML was used mainly because it is partially human readable, but also it can reflect hierarchical structures of data, which were used in the engine.

There are many XML parsers written in C++, which are categorized into two types, event-driven and document driven. With event-driven parsers, a callback is registered and called when a tag is opened or closed. With a document driven parser, the file is parsed and a hierarchical structure is generated which can then be traversed by the user of the parser. TinyXML is a document driven parser, and was used in the Duality Engine. TinyXML was used in preference over an event driven parser, such as Expat, as many callback methods would need to be generated with an event driven parser.

8.8 Conclusion

The Duality Engine was designed as a reusable graphics engine capable of rendering dynamic scenes. As the targeted user of the engine is programmers with some experience of graphics engines, the interface was kept simple, though the core managers can be accessed to provide additional functionality.

A simple entity system was implemented. The system updated all entities during each iteration of the game loop, and allowed direct method access. Direct method access may not be wanted in certain situations. An alternative to this would be messaging; this would decouple the entity classes, and minimize unwanted direct method access. Additionally, the entity behavior was hardcoded into each entity class. This engine could be improved in the future with the addition of entity scripting.

IX Testing

To gain an understanding of the performance of the Duality Engine on different hardware, simple benchmarking was performed. The benchmark was performed on four machines with hardware of varying feature levels.

Machine No 1.
GPU: NVidia GeForce G210
DXlevel: DX 10.1
CPU: Intel Core 2 Quad (2.33Ghz)
Ram: 4096MB
OS: Windows 7 Home Premium 64bit

MachineNo.	TestNo.	3D (anaglyph)	Shadows	Fullscreen	FrameRate (average)	Render Time (ms)
1	1	×	×	×	14.3	69.93
1	2	×	×	✓	9.8	102.04
1	3	×	✓	×	10.0	100.00
1	4	×	✓	✓	13.2	75.76
1	5	✓	×	×	7.2	138.88
1	6	✓	×	✓	7.2	138.88
1	7	✓	✓	×	6.0	166.67
1	8	✓	✓	✓	7.1	140.85

Machine No 2.
GPU: NVidia GTX 460
DXlevel: DX 11
CPU: Intel E8500 Core 2 Duo (3.16Ghz)
Ram: 4096MB
OS: Windows 7 Professional 64bit

MachineNo.	TestNo.	3D (anaglyph)	Shadows	Fullscreen	FrameRate (average)	Render Time (ms)
2	9	×	×	×	153.0	6.54
2	10	×	×	✓	135.1	7.4
2	11	×	✓	×	120.5	8.30
2	12	×	✓	✓	130.0	7.69
2	13	✓	×	×	82.6	12.11
2	14	✓	×	✓	78.2	12.79
2	15	✓	✓	×	68.5	14.60
2	16	✓	✓	✓	66.3	15.08

Machine No 3.
GPU: ATI Radeon HD 5770
DXlevel: DX 11
CPU: AMD Phenom II X3 720 (3.2Ghz)
Ram: 4096MB
OS: Windows 7 Professional 64bit

MachineNo.	TestNo.	3D (anaglyph)	Shadows	Fullscreen	FrameRate (average)	Render Time (ms)
3	17	✗	✗	✗	138.2	7.24
3	18	✗	✗	✓	135.4	7.39
3	19	✗	✓	✗	98.3	10.17
3	20	✗	✓	✓	91.0	10.99
3	21	✓	✗	✗	75.1	13.32
3	22	✓	✗	✓	72.2	13.85
3	23	✓	✓	✗	49.2	20.33
3	24	✓	✓	✓	44.5	22.47

Machine No 4.
GPU: NVidia GeForce 9800GT
DXlevel: DX 10
CPU: Intel E8760 Core 2 Duo (3.06Ghz)
Ram: 4096MB
OS: Windows 7 Home Premium 32bit

MachineNo.	TestNo.	3D (anaglyph)	Shadows	Fullscreen	FrameRate (average)	Render Time (ms)
4	25	✗	✗	✗	76.5	13.07
4	26	✗	✗	✓	85.9	11.64
4	27	✗	✓	✗	64.0	15.63
4	28	✗	✓	✓	61.1	16.37
4	29	✓	✗	✗	38.2	26.18
4	30	✓	✗	✓	42.3	23.64
4	31	✓	✓	✗	36.8	27.17
4	32	✓	✓	✓	36.5	27.40

X Evaluation

Upon reflection of the entire development of the project, the problems highlighted at the start of the project have been solved. The main problem of large quantities of dynamic lights has been solved with the efficient implementation of the light pre pass pipeline. In addition, stereoscopic rendering has been implemented with anaglyph filtering. Finally, the Duality Engine was developed as a graphics engine capable of rendering dynamic scenes. The engine is fully scalable, and can be extended upon in the future to support more advanced features, and optimisations.

Having developed the standard deferred shading pipeline in an early prototype, valuable knowledge was gained about the entire process of deferred shading. The techniques learnt to solve irregular issues from the prototype, were carried forward into the development of the main engine, such as the optimization of rendering point lights by replacing the quad with a much smaller sphere. In addition, by implementing the standard deferred shading pipeline, the material flexibility problems were highlighted early on, this allowed a newer alternative algorithm to be selected, the light-pre-pass pipeline. This solved the material flexibility problem. Developing the prototype also helped with the architectural design decisions for the Duality Engine, and focused the architecture around the batching system for efficiency.

Choosing DirectX11 as the primary graphics API, was a learning experience in itself. At the start of the project, the documentation on the API was minimal, provided almost no information on functions and classes. In addition, common DirectX9 features had been removed from the DirectX11 API, such as loading meshes, and font rendering. These challenges were overcome throughout the project, and developing a text rendering system has provided knowledge unforeseen at the start of the project.

Having completed the project, it’s apparent that the choice of DirectX11 over DirectX10 was immature, as no DirectX11 specific features were necessary in the implementation of the light pre pass pipeline or the stereoscopic rendering. Having said this however, the challenge was welcomed, and if the project could be restarted, DirectX11 would still be used again.

XI Bibliography

Akenine-Möller, T. Haines, E. and Hoffman, N., 2008. *Real-Time Rendering*. 3rd ed.
Massachusetts: AKPeters. Ch 7.5.3. Ch 7.9.2. Ch 9.1.

Brown, N., 2009. *Playstation®: Cutting Edge Techniques, Dynamic Resolution*
http://www.technology.scee.net/files/presentations/developliverpool/Cutting_Edge_Techniques_Develop_Liverpool_09.pdf
(Accessed 27th November 2010)

Deering, M., 1988. *The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics.*
http://Portal.acm.org/ft_gateway.cfm?id=378468&type=pdf
(Accessed 2nd December 2010)

Engel, W., 2009. Designing a Renderer for Multiple Lights: The Light Pre-Pass Renderer.
In Engel, W., ed. 2009. *ShaderX7*.
Massachusetts: Charles River Media. Ch 8.5.

Gumbau, J. Chover M. and Sbert, M., 2010. Screen Space Soft Shadows.
In Engel. W., ed. 2010. *GPU Pro*.
Massachusetts: AKPeters. Ch 4.1.

Hargreaves, S., 2004. *Deferred Shading.*
http://www.talula.demon.co.uk/DeferredShading.pdf
(Accessed 25th November 2010)

Kircher, S., Lawrance, A., 2009. *Inferred lighting.*
http://graphics.cs.uiuc.edu/~kircher/inferred/inferred_lighting_paper.pdf
(Accessed 1st December 2010)

Malan, H., 2009. Graphics Techniques in *Crackdown*.
In Engel, W., ed. 2009. *ShaderX7*.
Massachusetts: Charles River Media. Ch 2.6.

Pangerl, D., 2009. Deferred Rendering Transparency.
In Engel, W., ed. 2009. *ShaderX7*.
Massachusetts: Charles River Media. Ch 2.7.

Placeres, F.P., 2006. Fast Per-Pixel Lighting with Many Lights.
In Dickheiser, M., ed. 2006. *Game Programming Gems 6*.
Massachusetts: Charles River Media. Ch 5.7.

Placeres, F.P., 2007. Overcoming Deferred Shading Drawbacks.
In: Engel,W., ed. 2007. *ShaderX5*.
Massachusetts: Charles River Media. Ch 2.5.

Rabin, S., 2008. *Introduction To Games Development*
Massachusetts: Charles River Media. Ch 6.6. Ch 3.4.

Schüler, C., 2009. An Efficient and Physically Plausible Real-Time Shading Model.
In: Engel. W., ed 2009. *ShaderX7*.
Massachusetts: Charles River Media. Ch 2.5.

Trebilco, D., 2009. Light-Indexed Deferred Rendering.
In Engel, W., ed. 2009. *ShaderX7*.
Massachusetts: Charles River Media. Ch 2.9.

Valient, M., 2007. *Deferred Rendering in Killzone 2.*
http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf
(Accessed 12th October 2010)

VanVerth, J.M. Bishop, L.M., 2004. *Essential Mathematics for Games and Interactive Applications*. Massachusetts: Morgan Kaufmann Publishers. Ch 3.5.2.

Images:

Division of Tourism, 1996. *Monarch butterfly on Pentas flower at the Butterfly World attraction in Coconut Creek, Florida,* digital photograph. Accessed April 2011.
<http://ibistro.dos.state.fl.us/uhtbin/cgisirsi/x/x/0/5?library=PHOTO&item_type=PHOTOGRAPH&searchdata1=COM01215>

XII Appendices

Appendix A: Project Proposal

Department of Computing Degree Project Proposal

Name: Thomas Russell **Course:** Computer Games Development **Size:** double
Discussed with (lecturer): Laurent Noel **Type:** development

Previous and Current Modules
Mobile Computing

Problem Context
The standard forward rendering technique to render a 3D scene in real-time limits the number of dynamic lights to a fairly low number. In some cases however, it is required to have a large quantity of dynamic lights. For example, the headlights of moving cars would have an effect on the environment in a city scene at night. In conjunction to this, dynamic shadows cast from each individual light are not viable as a real-time solution with current technology. A deferred lighting approach will allow for multiple dynamic lights, however as the materials for each pixel are stored, this can lead to large memory usage.

The Problem
I intend to design a 3D graphics engine capable of supporting large quantities of dynamic lights. As traditional deferred shading suffers from a large memory usage, the light pre pass pipeline will be implemented.
Potential Ethical or Legal Issues
None

Specific Objectives

- Design the architecture of the system to support multiple dynamic lights.
- Develop an initial prototype to test feasibility of the project.
- Redesign the architecture of the system around newly acquired knowledge.
- Write a literature review discussing relevant literature
- Implement the new design.
- Write a report on the project.

The Approach

- The initial stage will be creating a prototype to see if the project is feasible. The prototype will use a simple deferred lighting technique of separating material properties for each pixel and calculating the effect of lights after all scene geometry has been processed.
- The architecture will then be redesigned around the knowledge acquired from development of the prototype. Issues may arise during development of the prototype which halt further development, so understanding them whilst at the initial prototype phase will help to understand how to solve these issues later on.
- The literature review will be written discussing relevant literature focussing on deferred shading techniques.
- The actual design will then be implemented, or extended from the prototype.
- The report will be written.

These key activities will be done in the order shown. Documentation will be kept along the way.

Resources
Microsoft Visual Studio 2008 / Microsoft Visual Studio 2010
Microsoft Office Visio 2007
Microsoft DirectX SDK
OpenOffice Writer

Potential Commercial Considerations
None

Estimated costs and benefits
The main cost factors of the project come from the amount of time the developer will be working on it. The key benefits are that the software will be able to render a 3D scene with multiple dynamic lights with a fair level of realism. The graphics engine could be used to create simulation, visual representation or video game software, at a lower cost than if a company were to purchase and use a license for an external 3D engine.

Literature Review Content
Deferred Shading Techniques

References
Akenine-Möller, T. Haines, E. & Hoffman, N., 2008. *Real-Time Rendering, 3rd ed*. AK Peters.

Nguyen, H., 2008. *GPU Gems 3*. Nvidia, Ch 19.

Pangerl, D., 2009. Deferred Rendering Transparency. In Engel,W., 2009. *ShaderX7: Advanced Rendering Techniques*. Course Technology.

Pharr, M., 2005. *GPU Gems 2*. Nvidia, Ch 9.

Thibieroz, N., 2009. Deferred Shading with Multisampling Anti-Aliasing in DirectX 10. In Engel,W., 2009. *ShaderX7: Advanced Rendering Techniques*. Course Technology.

Appendix B: Literature Review

Deferred Shading Techniques

Thomas Russell,
BSc (Hons) Computer Games Development

Project: Real-time scene rendering with a high number of dynamic lights.
Supervisor: Laurent Noel
Second Reader: Gareth Bellaby
21 April 11

Abstract

Decoupling the lighting calculations from the rendering of scene geometry using deferred shading, allows a dynamic scene to be rendered with full dynamic lighting in less time at the cost of higher memory usage.

By performing a lighting stage before material processing, the memory impact can be minimized, with the benefits of full dynamic lighting and flexible lighting models per material, at a slight cost of extra processing.

This extra processing can be minimized however, by using a lower resolution texture than the final render, and up-scaling whilst applying a smart edge-detection filter.

Introduction

1.1 Context

Rendering a scene with dynamic lighting in real-time is a core renderer feature for most modern real-time renderers. As pixel shading techniques increase in complexity, previous methods of calculating the dynamic light contribution on geometry are too slow for scenes with many lights due to pixel overdraw. In some cases, the light contribution is calculated by rendering the scene geometry in multiple passes and combining the lighting until the contribution from every light is calculated (Engel, 2009).

In standard forward shading, due to the lighting being coupled with the geometry, the average time taken to render a single frame is directly proportional to the number of lights affecting the geometry and the complexity of the geometry (Hargreaves, 2004).

For a scene with eight lights, a forward renderer would use a shader generated for that number of lights. Generating shaders for each material with each number of lights available soon suffers from combinatorial explosion. With twelve lights and four different material types, this would combine to be forty-eight different shaders for four materials. This does not also take into account different light types. This raises issues with architecting an optimized rendering system using batching, as this would require batching by the number of lights and light types to achieve a good batching system.

1.2 Overview

Using a deferred shading approach, a scene can be rendered in real-time with a high number of lights irrespective of scene geometry complexity (Valient, 2007). Section 2 presents the concept of the deferred shading approach, benefits of using deferred shading and known implementation issues. Section 3 presents three alternative solutions which develop the deferred shading concept.
Deferred Shading

2.1 The Deferred Shading Concept

The standard forward rendering technique to render a scene using dynamic pixel lighting involves calculating the most influential lights and applying lighting when rendering geometry (Placeres, 2006). A common approach for applying many lights is to apply different shaders for each material for different quantities and types of lights. Due to the tight coupling between geometry, materials and lighting calculations, scene complexity is proportional to the number of objects combined with the number of applied lights (Hargreaves, 2004).

Figure 2.1.1 An in-game night scene from GrandTheftAuto IV. All car headlamps and traffic lights are dynamic. Lighting is calculated using a deferred approach.

An issue with performing lighting calculations when rendering geometry arises when new geometry is rendered over existing geometry. This overdraw impact can be lowered using a deferred approach.

The deferred shading technique was first proposed by Deering (1988), though the technique wasn't labelled "Deferred shading" until later (Hargreaves, 2004). By storing the data required to complete the lighting equation, the lighting can be decoupled from the rendering of geometry, this allows the lighting to be applied as a post-process (Placeres, 2006).

The deferred shading concept is composed of three main stages: the *geometry* stage – where the scene is rendered and material data is stored, the *lighting* stage- where the lighting is calculated, and the *composition* stage – where the lighting is combined with the material from the geometry stage.

As the lighting calculation is removed from the geometry stage, the implication of overdraw is minimised, as lighting calculations won't be wasted on pixels which don't appear in the final render. This allows a higher number of lighting calculations per pixel than a standard forward renderer, and thus more lights.

2.2 The G-Buffer

The lighting stage requires the position data and normal data for each pixel to apply the lighting as a post-process. This data is gathered into a collection of textures during the geometry phase, known as the Geometry Buffer (or "*G-Buffer*") to allow the equation to be completed at a later stage (Akenine-Möller, Haines, and Hoffman, 2008).

Frank Puig Placeres (2007) proposed the G-Buffer structure shown in Figure 2.2.2 as an initial starting point for deferred shading. However, this structure does not store any material properties, such as the diffusive colour of the object, or how reflective the material is. Furthermore, the same lighting model must be applied to each pixel during the lighting stage (Engel, 2009).

For instance, if the Blinn-Phong lighting model was used to construct the lighting accumulation, the entire scene would be lit using Blinn-Phong shading. This may not be desired, as different lighting models are suited to different materials, i.e. Minneart for a silk dress.

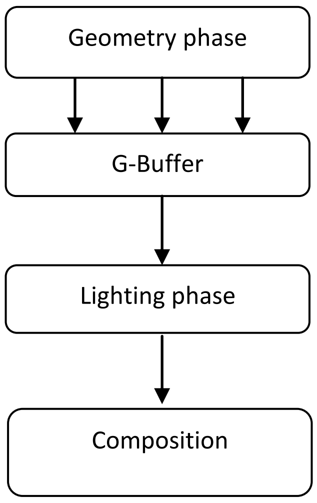


Figure 2.2.1
Deferred shading pipeline (courtesy of Frank Puig Placeres)

	R	G	B	A
Texture 1	Position X	Position Y	Position Z	[empty]
Texture 2	Normal X	Normal Y	Normal Z	[empty]

Figure 2.2.2 Example G-Buffer layout (courtesy of Frank Puig Placeres)

To solve this problem, a value could be stored in either the position or normal texture's alpha channel to indicate which lighting model to use, although these channels may be required to store material information as shown in Figure 2.2.3.

	R	G	B	A
Texture 1	Normal X	Normal Y	Normal Z	Scattering
Texture 2	Diffuse Colour R	Diffuse Colour G	Diffuse Colour B	Emissive
Texture 3	Specular Intensity	Specular Power	Occlusion	Shadow amount
Texture 4	Depth	Depth	Depth	Depth

Figure 2.2.3 Example G-Buffer layout (courtesy of Shawn Hargreaves)

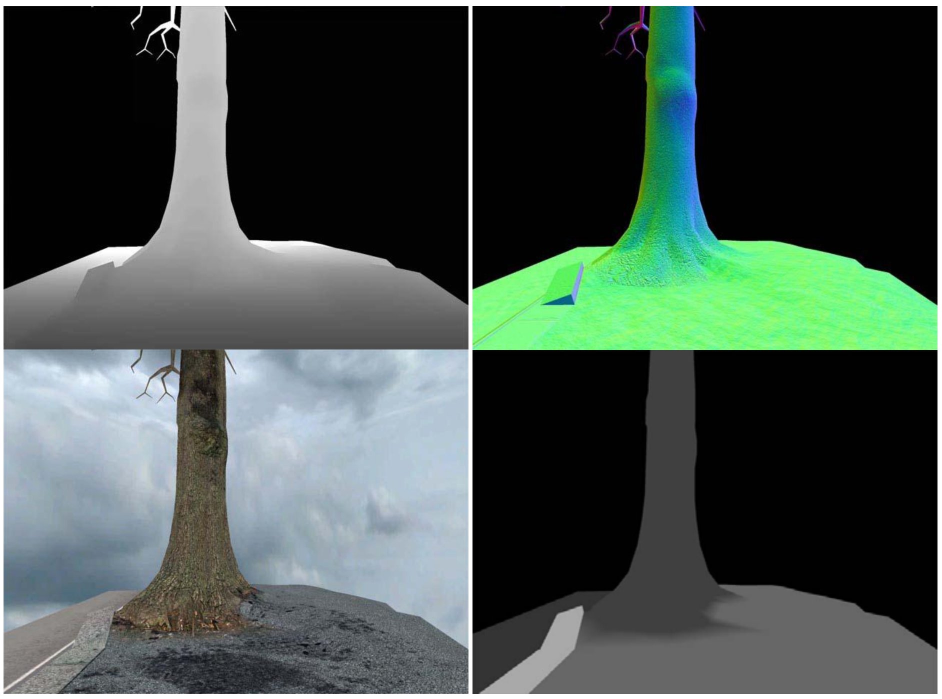


Figure 2.2.4 Example texture components of a G-Buffer; depth (upper left), normal (upper right), diffuse (lower left), specular intensity (lower right). (Courtesy of Shawn Hargreaves)

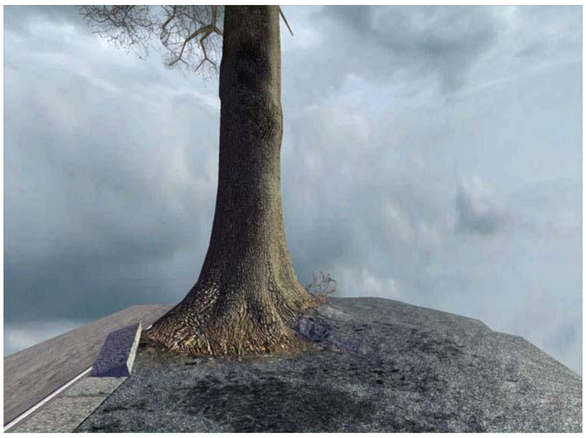


Figure 2.2.5
Example composition of lighting accumulation with G-Buffer textures.(Courtesy of Shawn Hargreaves)

To specify material attributes, the G-Buffer must contain more textures as demonstrated in Figure 2.2.3 and Figure 2.2.4 with the addition of diffusive colours, specular properties and other material attributes. These material attributes are then combined with the accumulated lighting to create the final frame image as shown in Figure 2.2.5.

The depth of each pixel can be stored rather than the position, as the viewspace and worldspace positions can be reconstructed using the depth (Hargreaves, 2004). This allows a single value to be stored instead of three, so more material data can be stored in the texture by lowering the precision of the depth value stored. Shown in Figure 2.2.3.

2.3 Issues with Deferred Shading

As the lighting is decoupled from the rendering of scene geometry, all material attributes which directly affect lighting must be output to the G-Buffer (Engel, 2009). As a result the flexibility of materials is tightly dependant on the available memory of the platform.

The G-Buffer is composed of several textures which are output from the geometry stage, so the platform must support Multiple Render Targets, otherwise the scene geometry must be processed for each texture (Engel, 2009). Aside this requirement, as each texel must be written for each texture in the G-Buffer, deferred shading often suffers from a high fill rate requirement. Where a standard forward approach would merely output a single colour value, a deferred approach would output four sets of values per pixel.

Due to the G-Buffer storing the data per-pixel, only the closest fragment will be stored in the G-Buffer, which results in semi-transparent objects overriding the geometry data behind (Pangerl, 2009). For this reason semi-transparent geometry is not stored when writing the G-Buffer.

2.4 The Benefits of Deferred Shading

The most noticeable benefit of deferred shading is the availability of high quantities of dynamic lights within a scene. Alongside this, popular post-processing techniques have the desired texture inputs from the G-Buffer to not require rendering the scene using multiple passes. Depth of field, ambient occlusion and fog can sample the G-Buffer textures without needing to gather new data. New graphics techniques such as soft shadows calculated in screen space also benefit from using these textures (Gumbau, Chover and Sbert, 2010).

The lighting stage of deferred shading can also be interlaced with an existing forward renderer given the renderer supports a depth only pre pass. This was illustrated by Malan (2009) during the production of Crackdown, where the lighting for street lamps and car headlights were applied after the scene had been processed.

Alternate Solutions

The standard deferred shading concept stores material data at the pixel level and then calculates the accumulated lighting for each pixel using the material data. However, if the lighting can be calculated prior to rendering the geometry, the amount of data needed to bridge the lighting and geometry stage is minimised.

3.1 Light Indexed Deferred Rendering

Damian Trebilco (2009) proposed a solution to separate the lighting and geometry rendering stages; similar to standard deferred shading, but with light accumulation calculated prior to geometry rendering.

Trebilco modified the standard deferred shading pipeline to include a geometry depth-prepass, where the depth buffer is filled. This is then sampled, and used to reconstruct the position in view space of each pixel.

The lighting stage then iterates through the lights, each of which have a unique index, and decides if the light affects that viewspace position. If the position is affected by a light, the index of that light is stored in a Light-Index buffer.

During the geometry rendering stage, the pixel stage samples the light-index buffer to detect which lights affected the geometry, and uses the light index to look-up the light properties in light data textures. The light data textures contain data such as; the position, colour and attenuation.

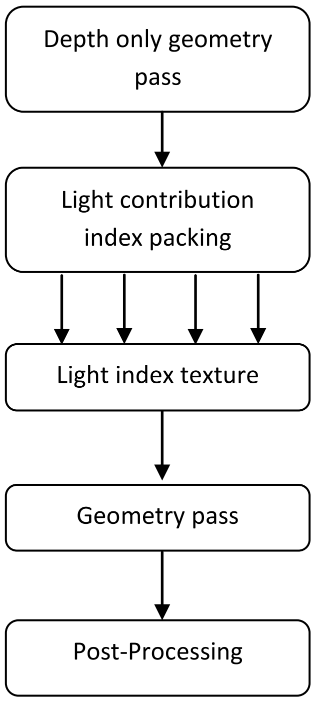


Figure 3.1.1
Example pipeline proposed by Damian Trebilco

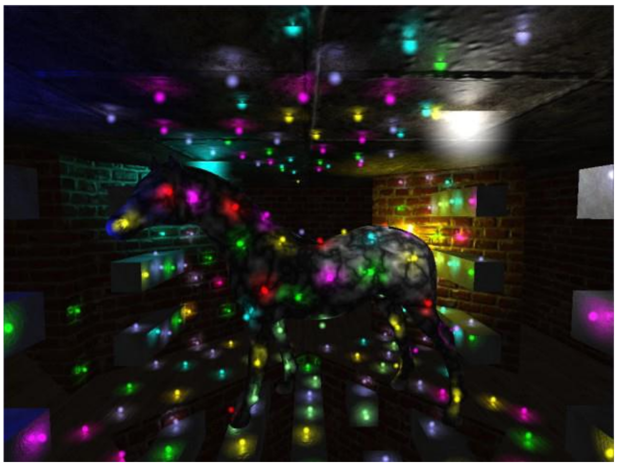


Figure 3.1.2
Example of Light Indexed Deferred Rendering
(Courtesy of Damian Trebilco)

Unfortunately, as the index is stored per pixel, the index can be overwritten by a later light which also affects the geometry.

(Trebilco 2009) derived a solution whereby the light index of four lights is packed into the texture's R,G,B and A channels, however this still means only four lights can contribute to a single pixel's lighting, unlike standard deferred shading where the number of effective lights per pixel is not bound by the storage method.

3.2 Light Pre-Pass deferred rendering

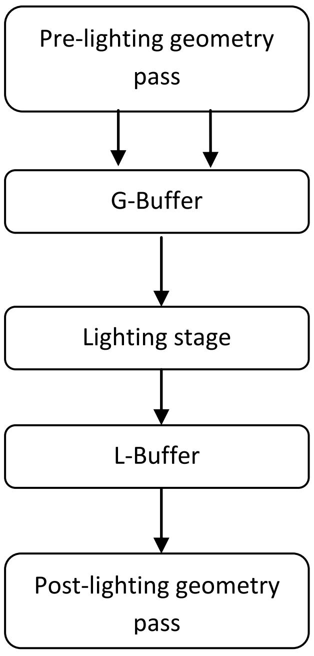


Figure 3.2.1
Example pipeline proposed by Wolfgang Engel

Wolfgang Engel (2009) introduced the concept of the light-pre-pass renderer as a solution to minimize the size of the G-Buffer. If lighting is calculated prior to geometry rendering, material data does not need to be stored, as this can be applied during the post-lighting geometry stage. This results in the light pre-pass concept having greater material flexibility than the standard deferred shading concept.

Where a standard deferred shading renderer would typically store the final light contribution for each pixel in the Light Buffer (*L-Buffer*) before combining the L-Buffer with the G-Buffer, Engel (2009) proposes storing light properties as terms of the lighting calculation. As this approach does not complete the equation, but outputs the lighting terms, the production of the L-Buffer requires less processing.

The original technique proposed by Engel (2009) only uses a single texture and does not store a specular component, as the lighting properties require four values to be stored independently. As the specular component should ideally be combined with the lighting buffer, an extra texture to store specular properties will be needed.

Applying the specular term to the fourth channel of the texture allows the diffuse and specular terms to be stored together, at the cost of the specular terms not modelling realistic specular lighting perfectly (Engel, 2009).

Figure 3.2.2 Light Pre Pass rendering used in Blur.

The light pre-pass concept allows the calculation of different lighting models to be evaluated at the post-lighting geometry stage and so allows further flexibility in material types. The concept could be adapted to store a closer approximation to the specular term, including specular light colour, however this would require six channels, three for diffusive properties and three for specular.

3.3 Inferred Lighting

Scott Kircher and Alan Lawrance (2009) present an extension of the Light Pre Pass concept. Utilizing the result of separating the lighting from the material and geometry stages, Kircher and Lawrance proposed that calculating lighting at a different resolution to the final render is possible with minimal visual artefacts. By calculating the lighting for a texture at 60% the size of the final rendered image, 40% of the lighting calculations need not be calculated. This optimization allows for more lights, or for the processing to be used elsewhere.

Unfortunately, during up-scaling the light texture when sampling at the final geometry stage, visual artefacts appear where the edges of rendered polygons lose definition.

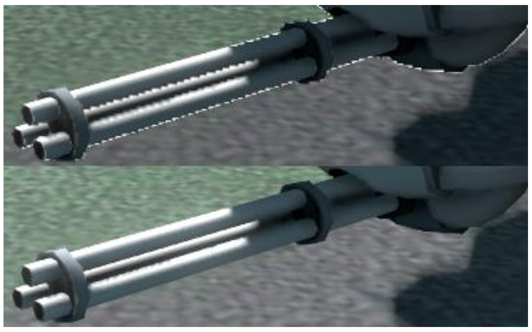


Figure 3.3.1
Example artefacts due to up-scaling the light buffer (Top).
Using the DSF to solve this issue (Bottom).
(Image courtesy of Kircher, S. and Lawrance, A.)

Scott Kircher and Alan Lawrance (2009) propose using a Discontinuity Sensitive-Filter (DSF) applied to the lower resolution image as an effective edge-detection between discontinuous polygons. This filter technique requires a unique object ID and polygon ID to be generated for the geometry and stored per vertex. The DSF decides if two pixels are edges if their IDs do not match.
When combined with the up-scaling of the L-Buffer, this solution solves many of the artefacts.

Inferred lighting has the potential to require less computation and thus perform faster than Light-Pre Pass implementations of the same resolution, as the resolution of the L-Buffer can be scaled down (Brown, 2009).

3.3.1 Transparency

Developing the work of David Pangerl(2009), lighting for four layers of semi-transparent geometry can be calculated using a stippled pattern when writing the geometry to the G-Buffer. This pattern can then be reversed at the final geometry stage to re-produce the four layers of transparency with lighting, using the DSF (Kircher, Lawrance, 2009). This is an elegant solution to the transparency problem deferred shading renderers typically suffer from.

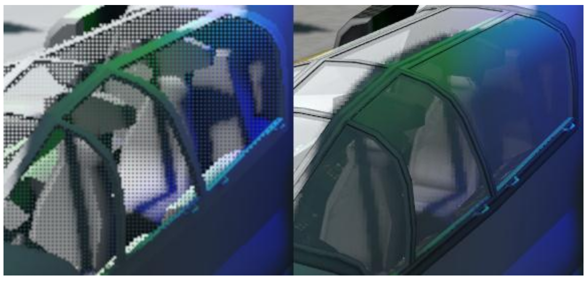


Figure 3.3.2
Example of stippled pattern when rendering semi-transparent geometry (Left), and the transparency solved using the DSF (Right).

Conclusion

Rendering a scene with many dynamic lights takes less processing time when using a modified deferred shading pipeline over standard forward rendering. Moreover, it's possible to increase flexibility of materials – and lighting models, by deferring lighting calculations to a later geometry pass, as shown with the Light-Pre-Pass and Light Indexed designs.

Due to the limitation of a light overlap boundary, a Light-Indexed deferred shading implementation may not provide enough lighting accuracy when a scene contains many overlapping lights; yet it will perform much faster than a standard forward approach for the same scene, and require less video memory than a standard deferred shading solution.

In a contrasting situation where the accumulated lighting must be accurate, and the target platform does not support multiple render targets, a Light-Pre-Pass implementation will perform faster than a standard deferred approach, with less visual errors where lights overlap.

Finally, by lowering the resolution of the light buffer and using a smart edge-detection filter, lighting calculations can be performed on fewer pixels and thus increase performance, at slight accuracy cost. This concept could equally be applied to the standard deferred approach, along with the Light-Indexed approach.

References

Akenine-Möller, T. Haines, E. and Hoffman, N., 2008. *Real-Time Rendering*. 3rd ed.
Massachusetts: AKPeters. Ch 7.9.2.

Brown, N., 2009. *Playstation®: Cutting Edge Techniques, Dynamic Resolution*
http://www.technology.scee.net/files/presentations/developliverpool/Cutting_Edge_Techniques_Develop_Liverpool_09.pdf
(Accessed 27th November 2010)

Deering, M., 1988. *The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics.*
http://Portal.acm.org/ft_gateway.cfm?id=378468&type=pdf
(Accessed 2nd December 2010)

Engel, W., 2009. Designing a Renderer for Multiple Lights: The Light Pre-Pass Renderer.
In Engel, W., ed. 2009. *ShaderX7.*
Massachusetts: Charles River Media. Ch 8.5.

Gumbau, J. Chover M. and Sbert, M., 2010. Screen Space Soft Shadows.
In Engel. W., ed. 2010. *GPU Pro*.
Massachusetts: AKPeters. Ch 4.1.

Hargreaves, S., 2004. *Deferred Shading.*
http://www.talula.demon.co.uk/DeferredShading.pdf
(Accessed 25th November 2010)

Kircher, S., Lawrance, A., 2009. *Inferred lighting.*
http://graphics.cs.uiuc.edu/~kircher/inferred/inferred_lighting_paper.pdf
(Accessed 1st December 2010)

Malan, H., 2009. Graphics Techniques in *Crackdown.*
In Engel, W., ed. 2009. *ShaderX7.*
Massachusetts: Charles River Media. Ch 2.6.

Pangerl, D., 2009. Deferred Rendering Transparency.
In Engel, W., ed. 2009. *ShaderX7.*
Massachusetts: Charles River Media. Ch 2.7.

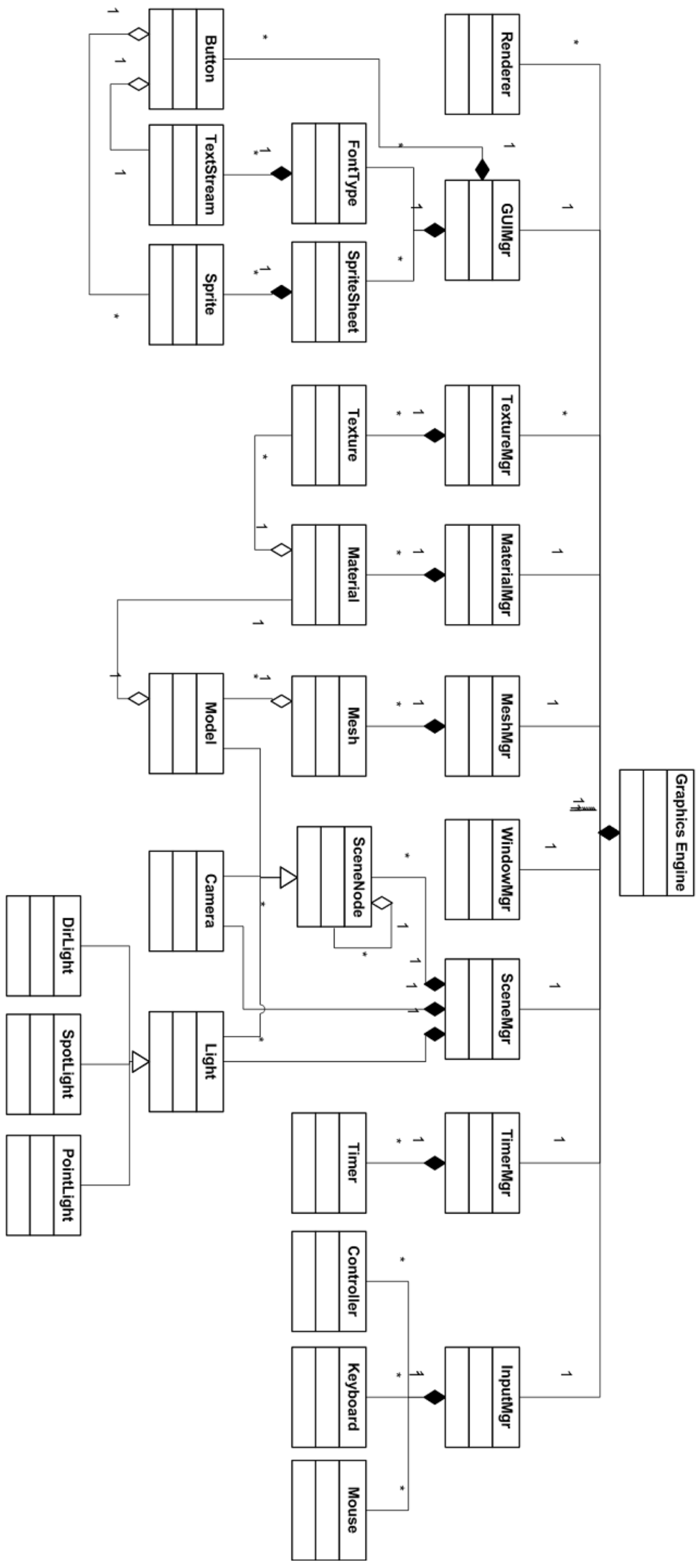
Placeres, F.P., 2006. Fast Per-Pixel Lighting with Many Lights.
In Dickheiser, M., ed. 2006. *Game Programming Gems 6*.
Massachusetts: Charles River Media. Ch 5.7.

Placeres, F.P., 2007. Overcoming Deferred Shading Drawbacks.
In: Engel,W., ed. 2007. *ShaderX5*.
Massachusetts: Charles River Media. Ch 2.5.

Trebilco, D., 2009. Light-Indexed Deferred Rendering.
In Engel, W., ed. 2009. *ShaderX7.*
Massachusetts: Charles River Media. Ch 2.9.

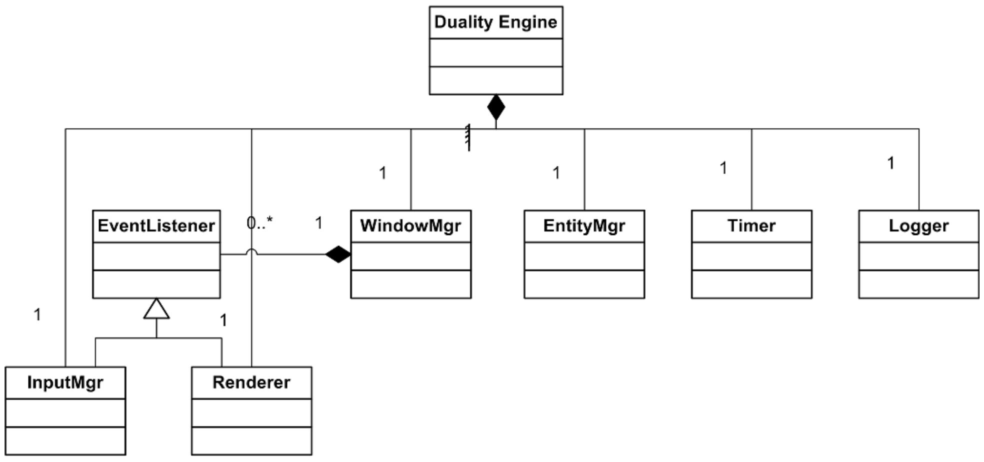
Valient, M., 2007. *Deferred Rendering in Killzone 2.*
http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf
(Accessed 12th October 2010)

Appendix C: Prototype UML

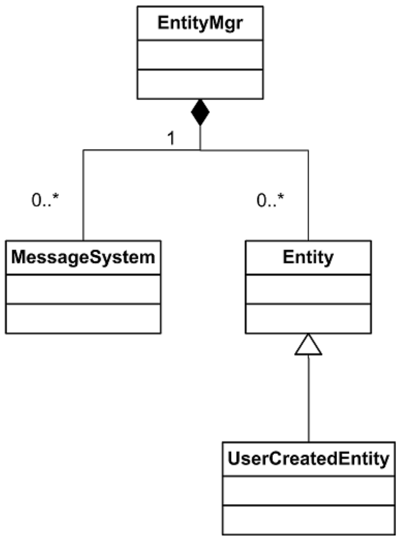


Appendix D: Duality Engine UML

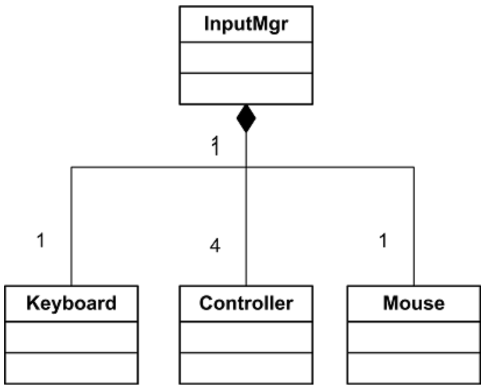
Duality Engine structure



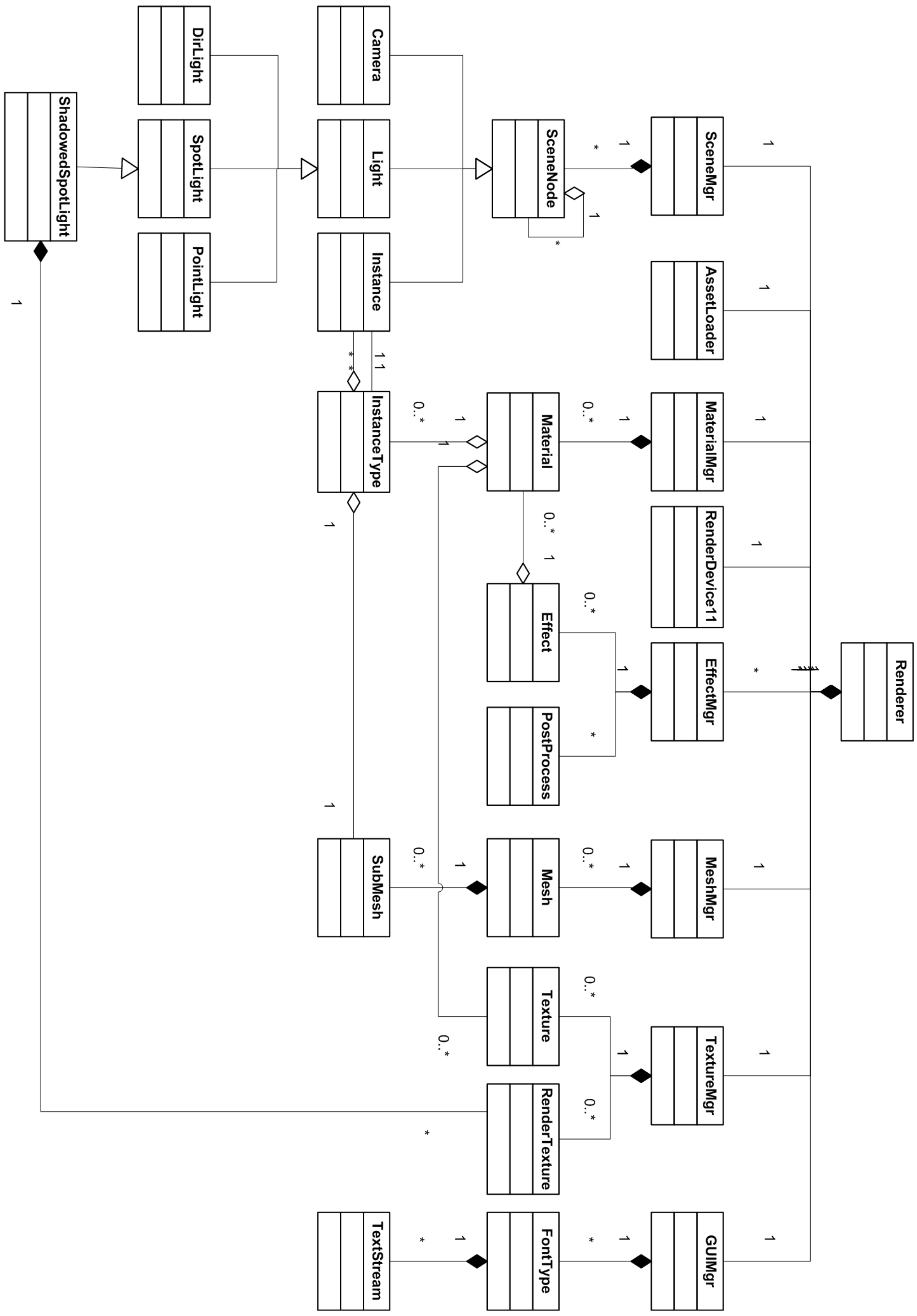
Entity Manager structure



Input Manager structure

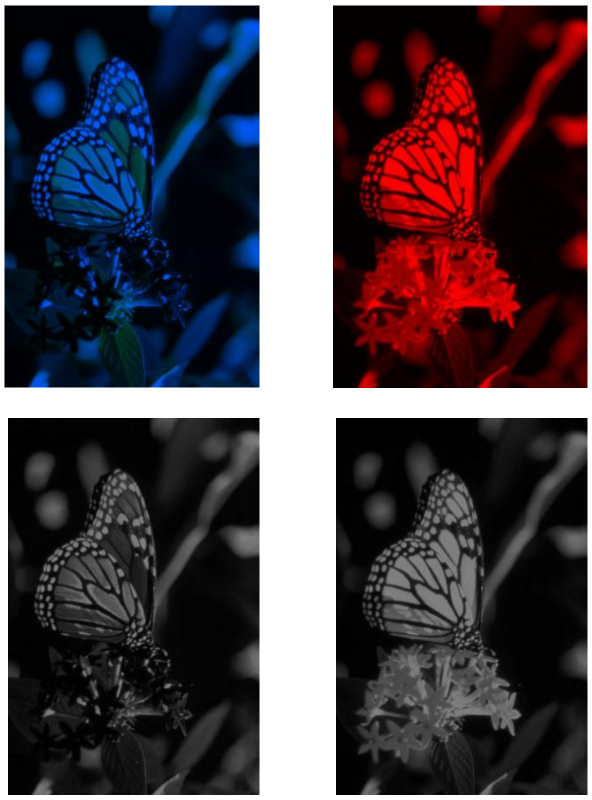


Renderer structure



Appendix E: Simple anaglyph issues

Frequency issues with using a simple implementation of anaglyph filtering. Top, cyan and red channels used for filtered image. Bottom, the filters removed, image as seen by receiver. Notice the difference in the brightness and contrast of the images.



Appendix F: Pure anaglyph

Frequency issues removed by saturating the images before applying the filters. Top, cyan and red channels used for filtered image. Bottom, the filters removed, image as seen by receiver. Notice how the images are now the same, minimizing retinal rivalry.