The IB Extended Essay

Candidate: Tristan Seifert

IB Class of 2016

Subject Area: Computer Science

Word Count: 3826 Words

Candidate Number: 123

Principal: Laurelyn Arterbury

IB Coordinator: Stephanie Childress

Essay Coordinator: Mr. Michael Quatro

Subject Supervisor: Mrs. Deborah Kariuki

Westwood High School: 000883

Steps Toward Achieving Photorealism in 3D Graphics

by

Tristan Seifert

Research Question: *What are some 3D rendering techniques that can be used to achieve photorealism, and their impact on visual quality and performance?*

Abstract

Primarily, the aim of this investigation is to to answer the research question: *What are some 3D rendering techniques that can be used to achieve photorealism, and their impact on visual quality and performance?* Before analyzing the extent to which photorealism can be attained, and the ways in which techniques that achieve said goal are implemented, there are several factors to take into account: firstly, the current state of real-time 3D graphics; secondly, the capability of existing hardware; and the extent to which quality can be sacrificed for speed.

This investigation makes use of a variety of sources, including various theses and academic papers; case studies of contemporary graphics programs; guides and documentation for 3D graphics application programming interfaces (APIs), as well as a multitude books detailing the precise implementation of a variety of graphics techniques, and various places for optimization.

Finally, this paper concludes with an optimistic outlook on the future of photorealism in 3D graphics—a belief corroborated by several facts. First, the computing power of Graphics Processing Units (GPUs) has been exponentially increasing, allowing for much more complex and accurate algorithms. Second, with the advent of a wide gamut of devices, all of which consumers expect to produce immersive 3D experiences—and their extreme range computing power, from small battery powered mobile devices, to extremely high-powered desktop computers with multiple discrete GPUs—it becomes extremely important to devise alternate approaches to simplify common rendering techniques to perform well on the weakest GPUs.

The future in which photorealistic real-time 3D graphics are achievable is not a distant one: in fact, it may have already arrived and found its place in everyday life, in the form of video games and interactive mobile applications.

Word Count: 284 Words

Contents

Introduction

1.1 A Short Introduction to Computer Graphics

Ever since computers have been able to output graphics in any form, developers have always sought to get better and more realistic graphics out of them. Compared to the blocky, low-resolution graphics of the 1970s, today's incredibly realistic 3D graphics have come incredibly far. However, this incredible realism comes at a cost: developers must balance precisely the graphical quality they desire with the capabilities of available hardware. *What are some 3D rendering techniques that can be used to achieve photorealism, and their impact on visual quality and performance?*

Many techniques and algorithms have been developed—most all of them in the last decade-and-a-half alone—that simplify complex natural phenomena in such a way that computers can efficiently approximate them, and approach much higher levels of photorealism.

The requirements of each project are different, but this investigation will look at several techniques commonly to most 3D visualizations (such as videogames,) and analyze what effect they have on the performance of the rendering pipeline, and compare the effects they have on the realism and appearance of the output.

1.2 Real-time 3D Graphics

While computers have been able to produce incredibly photorealistic 3D scenes for decades, it has only been recently that the computing capacity of GPUs has caught up and made it possible to do so in real time: a prime example of which are modern video games.

To be considered realtime, graphics must be rendered at interactive frame rates: rates at which the human eye is unable to discern the individual frames, and they flow together into one smooth image: similarly to how a film is many different frames shown in rapid succession.

Typically, 60 frames per second (fps) is used, as most displays' refresh rates are 60Hz—though 30fps and 120fps have been growing in popularity recently.

However, despite these advances in computing capacity, achieving both photorealism and interactive frame rates—the 'forbidden fruit' of 3D graphics—still remains an incredibly difficult task: a task which requires many clever techniques and trade-offs to simplify the complex interactions of objects in a 3D space.

1.2.1 Limiting Factors

In order to process a given scene and produce visual output, an immense amount of information needs to be rapidly accessed by the GPU, much of which is heavily processed by shaders and other elements of the graphics pipeline before it is even displayed.

Factors that limit the performance of a rendering technique can be divided into two groups:

Memory Access

The speed at which this information can be read from memory, known as the memory bandwidth, is often a limiting factor: an issue known as memory bus saturation. This is the most common type of bottleneck encountered in modern 3D graphics programming, due to the nature of the data stored and used, and how many GPUs have (comparatively) small memory busses.

Algorithm Complexity

Other times—particularly on lower-end GPUs—the compute units are fully utilized and cannot operate any faster, i.e. perform any more calculations in a given period of time, causing the rest of the graphics pipeline to stall: an unsurprising fact, considering that many effects rely on incredibly complex vector mathematics.

These pipeline stalls are among the most prohibitively expensive (in terms of cycles that could be spent performing calculations) operations a GPU can perform. It is therefore in the programmer's best interest to avoid them at all costs.

1.3 Analyzing Improvements

To analyze the effects of various techniques—deferred shading, high dynamic range (HDR), bloom, and fast approximate antialiasing (FXAA), a testbed with a flexible rendering pipeline that has features similar to those of a modern video game was developed. It is written entirely in standards-compliant `C99` and `C++1y`. (Excerpts of relevant code are provided in the Appendix.)

When coupled with debuggers, static and dynamic analyzers—such as Apple's Instruments and OpenGL Profiler—very accurate and fine-grained data about the impacts of these techniques on performance can be acquired, down to a breakdown of which line of code takes the longest to execute.

To maintain a constant environment, all tests will be run on the same machine: a mid-2012 MacBook Pro, featuring a 2.3GHz Intel i7, 16GB of RAM, and an NVIDIA GeForce GT 650M, with 512MB of video memory. Additionally, all tests are ran in windowed mode, at a resolution of 1024×768 pixels, 32 bits per pixel (bpp), using OpenGL version 4.2.

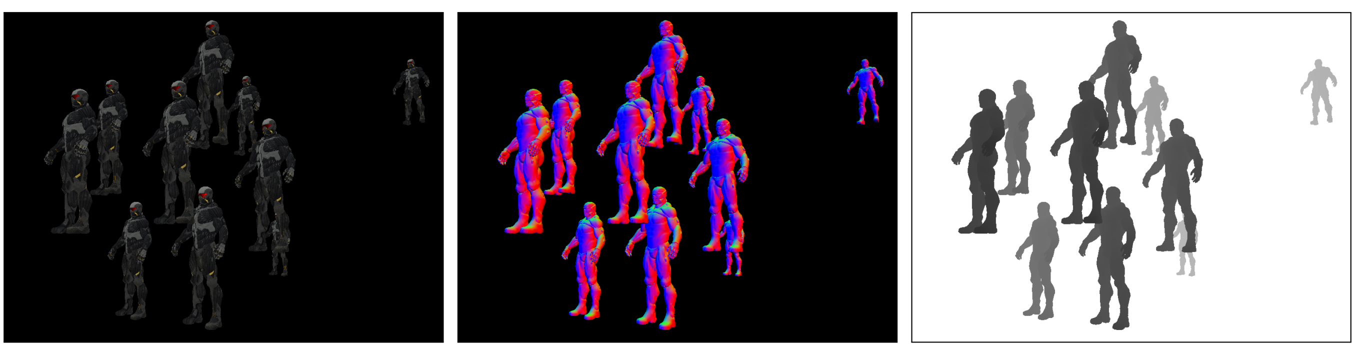
Deferred Shading

Traditionally, rendering pipelines calculated lighting information for every texel, regardless of whether it was visible in the final output, wasting an immense amount of resources, even if some invisible texels were discarded early on through depth tests.

Deferred rendering performs complex lighting calculations *after* all geometry has been rendered, freeing up significant compute resources for other tasks[1]. More realistic lighting can be implemented: for example, by using a higher exponent (thus leading to smoother reflections) with the Blinn-Phong reflection model, more accurately simulating the way lights interact.[2]

To achieve all of this, deferred shading works by rendering all texels into a geometry buffer, then running an additional shading pass to perform lighting calculations all at once.[3]

2.1 Geometry Buffer



(a) Albedo and specular (b) Surface normals (c) Depth

Figure 2.1: Components of the G buffer in the testbed's deferred shading implementation.

[1]Gábor Liktor and Carsten Dachsbacher. In: *GPU Pro 4: Advanced Rendering Techniques*. Ed. by Wolfgang Engel. CRC Press, 2013. Chap. Decoupled Deferred Shading on the GPU, pp. 81–97. ISBN: 978-1-4665-6744-3.

[2]Michal Ferko. "Real-time Lighting Effects using Deferred Shading". 2012. URL: `http://www.cescg.org/CESCG-2012/papers/Ferko-Real-time_Lighting_Effects_using_Deferred_Shading.pdf`.

[3]See Appendix A, `shader/lighting.shader`.

The geometry buffer is actually a collection of three distinct buffers: specular highlights (colours of sampled textures) and albedo; surface normals (used in calculating reflections) and fragment depth. World position is also required to perform lighting calculations, but it can be derived from depth information (and world-space transform matrices) via some vector mathematics.

Rendering of objects in the scene is performed by a comparatively simple shader[4], which serves to consolidate its inputs, mixing them as appropriate, and writing them to the G buffer. While deferred shading does require a significant amount of additional video memory—particularly since the normal and depth buffers need high precision floats to accurately represent their values—it simplifies the lighting processing immensely.

Information about lights, encoded in memory as uniform structures, is sent to a shader, which also takes the G buffer as an input. It performs the necessary calculations for each texel and outputs it to the next stage in the rendering pipeline.

2.2 Lighting Calculations

Thanks to the flexibility afforded by performing all lighting calculations simultaneously, many different types of lighting can be implemented. In the example, four types of lighting are supported: ambient light, directional lights, point lights, and spotlights. Each of these lights has an associated specular and diffuse colour, among other properties that control its appearance and effect on the scene.

2.2.1 Ambient Light

Ambient light is an average of all non-specific light sources in a scene, with a fixed colour, affecting every texel equally. This is typically used to model the way in which thousands of distinct light sources interact to produce the (approximately) same illumination level throughout the scene.

2.2.2 Directional Light

Directional lights are an approximation of light sources that are infinitely far away, modeled as a series of parallel light rays. They have a diffuse and specular colour, and a direction that indicates which way the light rays will be cast. Any texel that intersects a light ray from such a light will be affected by it.

[4] See Appendix A, `shader/model.shader`.

Directional lights are typically the only lights that will cause shadows to be cast, and as thus, there are very few of them in any given scene: each instance would require a separate shadow mapping pass to create accurate shadows.

2.2.3 Point Light

Point lights are similar to directional lights; but rather than a direction, they instead have a specific position. Light rays are cast in all directions (forming a sphere of a specified radius) from this center point, and like directional lights, any texel that intersects a light ray will be affected by it. Instead of a constant strength, however, the influence of a point light gets weaker the further the texel is from the light source.[5]

This attenuation is defined by a constant (K_c), a linear (K_l) and a quadratic (K_q) term, as well as the distance from the light (d) and unattenuated intensity (I):

$$F_{att} = \frac{I}{K_c + K_l * d + K_q * d^2} \tag{2.1}$$

To get the amount of light contributed by a point light towards the final output colour of a texel, its specular and diffuse colours are multiplied by F_{att} before mixing. Typically, K_c is 1.

2.2.4 Spotlight

Spotlights are special cases of point lights, with a direction in addition to a position, as well as a radius. They cast light rays as a cone with a given radius, and illuminate everything that falls within this cone. Toward the edge of the radius, the intensity of the light begins to rapidly decay.

2.3 End Result

When combining all of these types of lighting, a good approximation of a complex environment with many lights can be created. Adjusting light properties on-the-fly can help give the illusion of a more photorealistic environment.

But what about particularly bright lights, or very dark ones? The standard approach to rendering them will cause texels illuminated by them to appear as solid bright or dark areas, losing most detail that

[5] Dave Shreiner et al. In: *OpenGL Programming Guide*. 8th ed. Addison-Wesley, 2013. Chap. Light and Shadow, pp. 368–370. ISBN: 978-0-321-77303-6.

Figure 2.2: Output of the lighting stage, prior to gamma compensation, with brightness values of > 1.0 clipped. A skybox was rendered in areas where lighting calculations produced no output.

may have been present before. HDR solves that problem, by having the lighting pass output an intermediate, unbounded representation of light intensity.[6]

A downside of deferred shading is that objects that previously affected light in unique ways are more difficult to implement, since all lighting calculations are in the same shader. For example, it becomes very difficult to simulate the way in which a glass object might distort objects behind it, or how objects behind a flame are blurred from the heat rising from it.

Additionally, a significant amount of memory is consumed by the geometry buffer as compared with regular forward shading, so it is not uncommon to reuse components of the geometry buffer for other purposes later in the rendering pipeline.

[6]Damian Trebilco. *Light Indexed Deferred Lighting.* 2007. URL: `https://github.com/dtrebilco/lightindexed-deferredrender/raw/master/LightIndexedDeferredLighting1.1.pdf`.

2.3.1 Performance Impact

By far, deferred shading consumes a significant piece of the frame's processing time. Approximately 9.6% of the frame processing time goes towards deferred shading. However, compared to the tremendous savings over per-pixel shading, this is still a massive reduction in processing time.

Before deferred shading, lighting was calculated for each texel. Average frame rendering times sat at approximately 6.3mS, with rendering and lighting consuming a whopping 93.7% of that time—the majority of which was spent performing complex lighting calculations over texels that would not even be rendered. Implementing deferred shading single-handedly brought the per-frame rendering time down to 2.9mS.

Running Time		Self (ms)	Symbol Name
574.0ms	9.6%	3.0	gfx::SceneLighting::render()
356.0ms	5.9%	2.0	gfx::SceneLighting::sendLightsToShader()
89.0ms	1.4%	1.0	gl::glDrawArrays(gl::GLenum, int, int)
60.0ms	1.0%	0.0	gfx::SceneLighting::renderSkybox()
20.0ms	0.3%	2.0	glm::tmat4x4<float>::inverse
13.0ms	0.2%	0.0	gfx::ShaderProgram::setUniform1f(std::string, float)
12.0ms	0.2%	1.0	gfx::ShaderProgram::bind()
5.0ms	0.0%	1.0	gfx::Texture2D::bind()
4.0ms	0.0%	0.0	gfx::Texture2D::unbind()
3.0ms	0.0%	0.0	gfx::ShaderProgram::setUniformVec(std::string, glm::tvec3<float>)
2.0ms	0.0%	0.0	<Unknown Address>
2.0ms	0.0%	0.0	gfx::ShaderProgram::setUniformMatrix(std::string, glm::tmat4x4<float>)
1.0ms	0.0%	0.0	gfx::VertexArray::unbind()

Table 2.1: Stack trace showing computational impact of deferred shading.

Of the processing overhead incurred by deferred shading, approximately two thirds go towards sending lighting information (such as position, coefficients, colours, etc.) to the lighting shader. Another 10% go towards rendering a skybox.

As far as memory use goes, deferred shading is relatively resource hungry. Because of the amount of data that is needed for lighting, the geometry buffer becomes quite large. Three buffers need to be allocated, at full screen resolution—the albedo and specular buffer, a 32 bpp integer buffer; the surface normals, a 64 bpp floating point buffer; and the depth buffer, a combined depth and stencil floating point format, requiring 32 bpp.

An additional 14MB of memory are required for the geometry buffer. However, all components of the buffer can be reused after the lighting pass for other steps of the rendering pipeline to reduce the overall memory footprint.

High Dynamic Range (HDR) and Bloom

One of the biggest problems of computer graphics has traditionally been to approach the dynamic range of the human eye, because all display devices have a limited colour palette they can reliably and accurately display—thus leading to many 'almost convincing' images. Particularly, the human eye is much better at recovering detail from very bright and very dark areas than a computer display can show, so a large range of brightness values must somehow be mapped onto the handful of nonlinear brightness values displayed by computer monitors: this is exactly what HDR does, via a process called tone mapping.

Additionally, due to optical imperfections in lenses (the eye is really one big lens) there often appears bleeding of light from very bright to darker areas. By applying bloom, the brightness of a light source can be exaggerated and shown more clearly.

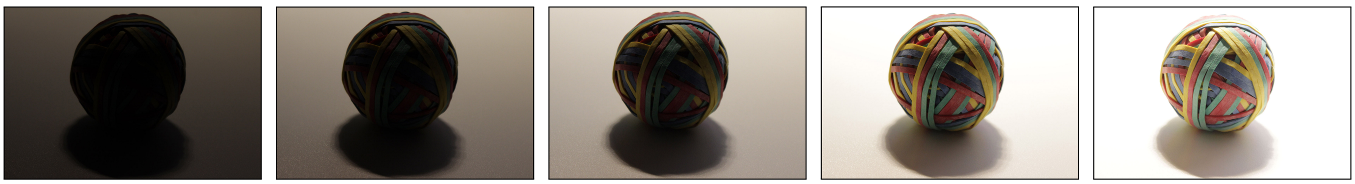
3.1 Producing HDR Output

(a) Before (b) After

Figure 3.1: Before and after tone mapping, gamma and white point adjustments.

The deferred shading pass outputs colour values into a floating point buffer, allowing for a nearly infinite amount of brightness values to be expressed. The conversion between HDR values and red, green, blue (RGB) values is relatively straightforward, and is defined by a function in a shader; also known as tone mapping. These values are also adjusted to match a certain white point[7].

In addition to a particular tone mapping algorithm, the sensitivity of HDR can easily be adjusted via an exposure parameter. This parameter serves as a constant multiplier for the HDR input colours before tone mapping, and affects the overall brightness of the image: similarly to how changing the exposure settings on a photographic camera will affect the brightness of the resultant image.



(a) -3 (b) -1.5 (c) 0 (d) 1.5 (e) 3

Figure 3.2: Effects of varying exposure values (EV) on a photograph.

Overall, HDR can produce a great improvement in visual quality with little additional work. An extra stage of shader processing before output adds minimal overhead, and no additional memory is needed, if a buffer from a previous stage in the rendering pipeline can be reused.

Various effects can be achieved simply by varying the exposure value: for example, a higher exposure value could be used for night-time scenes, and a lower one for day-time scenes. Automatic exposure adjustment, where the overall brightness is analyzed, and exposure is slowly changed to maintain a baseline level of brightness, similar to how the human eye functions—temporary blindness when going from a dark scene to a bright scene, or vice-versa, while the eye slowly adjusts to the difference in brightness—could be implemented, improving photorealism in interactive applications.

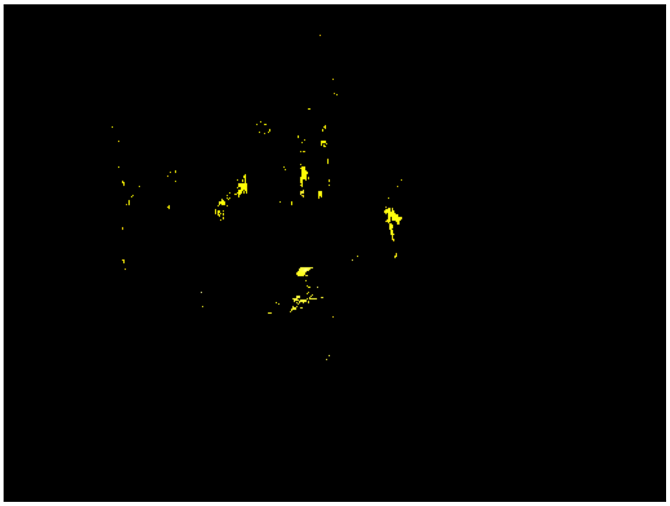
3.2 Bloom

Bloom simulates the glow that occurs around extremely bright light sources. In conjunction with HDR, it is easy to implement, incurring little additional overhead—but significantly increases photorealism, if implemented properly.

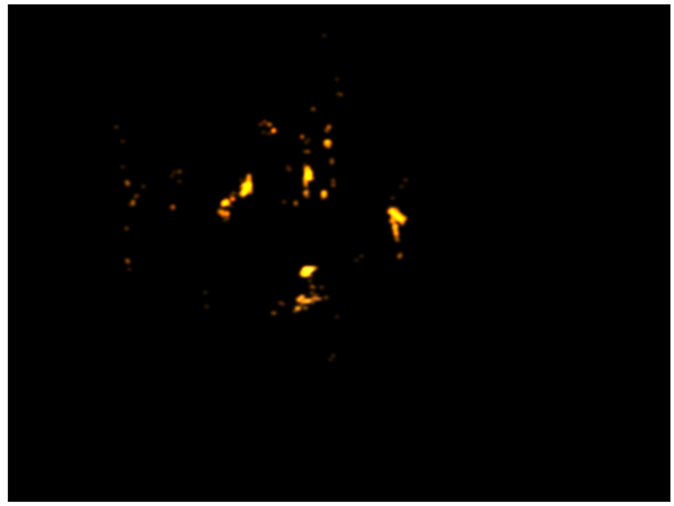
All fragments that are considered bright—a combined brightness of `1.0` or above—are copied into an additional buffer. This buffer is half the size of the output screen, thus saving memory and processing time.

[7] Fabien Houlmann and Stéphane Metz. “High Dynamic Range Rendering in OpenGL”. 2012.

This buffer is then blurred through several iterations of a Gaussian blur, until an adequate blur has been achieved.[8]



(a) Before



(b) After

Figure 3.3: Bright input fragments, before and after application of the Gaussian blur.

The Gaussian blur itself consists of a 13×13 blur kernel, which is approximated by sampling the texture seven times for each texel, using bilinear interpolation to simulate the effect of sampling many more times. It has been decomposed into separate horizontal and vertical components for performance reasons[9]—a performance impact of $O(n)$ rather than $O(n^2)$—and is run over a set of two buffers: the original 'bright fragment' input buffer, and the eventual 'output' buffer a predetermined number of times. Once blurring is completed, the output buffer is sampled in the HDR output shader, multiplied by a coefficient (determining the strength and effect of the blur on the rest of the scene,) then added to the calculated HDR colour.

3.3 End Result

By combining both HDR and blooming, the lighting in a scene already appears much more realistic, solving the issue of washed out highlights and details that disappear into the shadows. These two render passes require little in the way of additional memory, and their shader programs[10] are relatively simple, compared to various other techniques.

[8] Tiago Sousa, Nickolay Kasyan, and Nicolas Schulz. In: *GPU Pro 3: Advanced Rendering Techniques*. Ed. by Wolfgang Engel. CRC Press, 2012. Chap. CryENGINE 3: Three Years of Work in Review, p. 160. ISBN: 978-1-4398-8794-3.

[9] Filip S. *An investigation of fast real-time GPU-based image blur algorithms*. `https://software.intel.com/en-us/blogs/2014/07/15/an-investigation-of-fast-real-time-gpu-based-image-blur-algorithms`. Blog. 2014.

[10] See Appendix A, `shader/hdr.shader` and `shader/bloom.shader`.

Additionally, the HDR pass serves as a place for gamma correction to take place. Textures are stored as sRGB, and OpenGL's built-in conversion is disabled. This way, the user can configure the gamma of the application to match their monitor most closely, instead of relying on a hard-coded value in a graphics driver, or relying on the operating system to get it right.

3.3.1 Performance Impact

When analyzing the testbed's performance with HDR and blooming enabled, the relatively insignificant additional overhead incurred by the technique immediately becomes clear.

Running Time		Self (ms)	Symbol Name
HDR			
103.0ms	1.7%	0.0	gfx::HDRRenderer::render()
92.0ms	1.5%	0.0	gl::glDrawArrays(gl::GLenum, int, int)
5.0ms	0.0%	0.0	gfx::ShaderProgram::bind()
3.0ms	0.0%	0.0	gfx::VertexArray::bind()
2.0ms	0.0%	0.0	gfx::Texture2D::bind()
Bloom			
127.0ms	2.1%	0.0	gfx::BloomRenderer::render()
103.0ms	1.7%	0.0	gl::glDrawArrays(gl::GLenum, int, int)
10.0ms	0.1%	1.0	gfx::ShaderProgram::bind()
10.0ms	0.1%	0.0	gfx::ShaderProgram::setUniform1i(std::string, int)
2.0ms	0.0%	0.0	gfx::ShaderProgram::setUniform1f(std::string, float)
2.0ms	0.0%	0.0	gfx::ShaderProgram::setUniformVec(std::string, glm::tvec3<float>)

Table 3.1: Stack trace showing computational impact of HDR and blooming.

After optimizing the HDR code to reduce pipeline stalls, on average, 1.7% of a frame's processing time was taken up by the blurring of highlights (blooming,) and tone mapping of the output.

In addition, 6.2MB of video memory were needed for buffers. The two quarter resolution buffers for blooming are 48 bpp floating point buffers without alpha components, while the input buffer to the HDR process is a 64 bpp floating point buffer.

Considering the improvement in visual quality that a properly implemented HDR approach can give, the additional performance overhead is almost negligible. However, the difficulty lies in determining and implementing a good tone mapping algorithm that gives an output that looks realistic—not too saturated, but not too bland, either.

Fast Approximate Antialiasing (FXAA)

Due to the limited resolution of textures and other data, as well as the multitude of transformations applied to geometric primitives, it is extremely common for an unprocessed render output to exhibit heavy aliasing. Often, aliasing takes the form of jagged edges, but it can also manifest itself as strange and unnatural transitions between colours, contributing negatively toward the quality of the rendered image.

In the past, aliasing was combatted by rendering the entire scene at a much higher resolution—often 2x or 4x larger than the physical display resolution—then simply downscaling it, creating a primitive form of supersampling. Later on, similar techniques were applied to shaders, causing multisampling to take place when they sampled textures, not when they produced their output, improving performance somewhat; also known as MSAA. Supersampling works in a similar manner.

What all of these antialiasing algorithms have in common is that they are very computationally expensive. They can double or quadruple the rendering time, while yielding a minimal benefit.

4.1 Implementation

Processing the output with FXAA is straightforward. A buffer is created, into which the output of all previous stages of the rendering pipeline is stored, instead of directly going to the window framebuffer. This buffer is then set as an input to the FXAA shader program, which samples it, detects edges, smoothes them, and outputs a final antialiased output to the window framebuffer[11].

Because FXAA is implemented in a shader[12], rather than in hardware, its behavior (such as edge detection sensitivity, smoothing algorithm and sharpness, etc.) can be adjusted on-the-fly by the user to produce the quality of output they desire.

[11] Timothy Lottes. *FXAA*. 2009. URL: `http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf`.

[12] Example implementation from NVIDIA, Version 3.11 by Timothy Lottes

(a) With FXAA

(b) Without FXAA

Figure 4.1: A crop from the final output, with and without FXAA. Note the rough edges on on (b).

4.2 End Result

By utilizing a newly developed algorithm to approximate antialiasing instead of wasting precious computational resources and memory bandwidth on traditional algorithms, immense performance gains can be had. In most cases, the quality of FXAA is comparable to that of more traditional antialiasing algorithms: and most of the time, the precise nature of the antialiasing algorithm makes little difference to the user of the program, so long as aliasing artifacts are minimized.

4.2.1 Performance Impact

Implementing FXAA improves the quality of the output significantly, by removing aliasing artifacts, with little impact on the performance of the rendering pipeline.

Running Time		Self (ms)	Symbol Name
127.0ms	2.1%	0.0	`gfx::FXAARenderer::render()`
101.0ms	1.6%	0.0	`gl::glDrawArrays(gl::GLenum, int, int)`
10.0ms	0.1%	2.0	`gfx::ShaderProgram::setUniform1f(std::string, float)`
10.0ms	0.1%	0.0	`gfx::ShaderProgram::setUniform1i(std::string, int)`
5.0ms	0.0%	0.0	`gfx::ShaderProgram::bind()`
1.0ms	0.0%	0.0	`gfx::Texture2D::bind()`

Table 4.1: Stack trace showing computational impact of deferred shading.

On average, running the FXAA pass, using the highest 'low dither' preset specified by the shader, 2.1% of processing time is required to execute the FXAA algorithm each frame. The majority of this time is spent in the OpenGL library, waiting for the GPU to be ready to accept a command, so there is room for further optimization.

In addition, another colour buffer is needed, increasing memory overhead by approximately 4MB. (A simple 24 bpp buffer without alpha will suffice for this application, since all HDR processing will have already been done on the more complex buffers.)

For the improvement in visual quality—in particular, when using large output displays, and the low impact on performance—the impact of implementing FXAA is basically nil.

However, it is important to consider the impact that FXAA might have on various graphical overlays, such as HUDs, or various user interface elements. Often times, unexpected artifacts are created when FXAA works on such overlays, so they would be rendered in another buffer, and then overlaid on the output of the FXAA shader.

Overall Performance

To determine the overall effect on performance of the aforementioned rendering techniques, Apple's Instruments software was used to capture stack traces, and combining those with timing information within the testbed itself.

Running Time		Self (ms)	Symbol Name
5956.0ms	100.0%	3.0	`gfx::StandardRenderer::render()`
4021.0ms	67.5%	5.0	`gfx::SceneRenderer::render()`
574.0ms	9.6%	3.0	`gfx::SceneLighting::render()`
531.0ms	8.9%	0.0	`gfx::BloomRenderer::beforeRender()`
205.0ms	3.4%	0.0	`gfx::SceneRenderer::beforeRender()`
127.0ms	2.1%	0.0	`gfx::BloomRenderer::render()`
127.0ms	2.1%	0.0	`gfx::FXAARenderer::render()`
103.0ms	1.7%	0.0	`gfx::HDRRenderer::render()`
96.0ms	1.6%	0.0	`gfx::SceneLighting::beforeRender()`
57.0ms	0.9%	0.0	`gfx::SceneRenderer::renderNormally()`
43.0ms	0.7%	1.0	`gfx::SceneLighting::bindGBuffer()`
19.0ms	0.3%	0.0	`gfx::LevelCamera::updateViewMatrix()`
18.0ms	0.3%	0.0	<Unknown Address>
8.0ms	0.1%	0.0	`gfx::HDRRenderer::bindHDRBuffer()`
3.0ms	0.0%	0.0	`gfx::BloomRenderer::bindBloomBuffer()`
3.0ms	0.0%	1.0	`gfx::FXAARenderer::beforeRender()`
2.0ms	0.0%	0.0	`gfx::FrameBuffer::unbindRW()`
2.0ms	0.0%	0.0	`gfx::HDRRenderer::beforeRender()`
2.0ms	0.0%	0.0	`gfx::LevelCamera::getViewMatrix()`
1.0ms	0.0%	1.0	`glm::detail::tmat4x4<float>::tmat4x4()`
1.0ms	0.0%	0.0	`gfx::LevelCamera::getCameraLookAt()`
1.0ms	0.0%	0.0	`gfx::SceneLighting::setFXAA(gfx::FXAARenderer*)`
1.0ms	0.0%	0.0	`gfx::LevelCamera::getCameraPosition()`
1.0ms	0.0%	0.0	`gfx::SceneRenderer::afterRender()`
1.0ms	0.0%	0.0	`gfx::FXAARenderer::bindFXAABuffer()`
1.0ms	0.0%	0.0	`gfx::SceneLighting::afterRender()`

Table 5.1: Stack trace showing each step of the rendering pipeline, after running it for 60 seconds.

The implementation of these techniques does indeed affect the quality of the output in a positive way. Deferred shading allows for many more lights, giving a more realistic model of how light interacts with objects in the real world. HDR and blooming more accurately model the way in which the human eye perceives light, matching its dynamic range, and emulating some of its imperfections. FXAA smoothes the final output, getting rid of unpleasant rendering artifacts without the excessive overhead of previous methods antialiasing or supersampling.

However, these gains in appearance are not free in any sense of the word. While deferred shading reduces processing time required to calculate lighting for each texel significantly, it requires a significant amount of memory to store the additional data required to later render the lighting effects. Additionally, a plethora of lighting data must be sent to the shader every frame.

HDR and FXAA require additional memory as well, but provide an incredible increase in the quality of the output with little additional processing time required.

While these basic techniques by themselves do not magically create photorealistic outputs, they are significant steps towards allowing for much more complex shaders, and other processing techniques, that allow true photorealism.

For example, dynamic reflections and shadows could be used to enhance the appearance of objects. More complex and detailed textures and normal maps could allow rendered objects to have more detail than the mesh used to render them really does. Together, these techniques serve as an important foundation for more complex techniques working towards photorealism.

An effective base of low-overhead techniques that can be extended at a later time are necessary for more advanced techniques—such as ambient occlusion or 3D shadow mapping—to build on.

Soon, the smartphones many carry in their pockets might be able to provide immersive 3D experiences, all while maintaining impressive battery life and rendering stunning graphics on their high pixel density displays.

Bibliography

Ferko, Michal. “Real-time Lighting Effects using Deferred Shading”. 2012. URL: http://www.cescg.org/CESCG-2012/papers/Ferko-Real-time_Lighting_Effects_using_Deferred_Shading.pdf.

Houlmann, Fabien and Stéphane Metz. “High Dynamic Range Rendering in OpenGL”. 2012.

Liktor, Gábor and Carsten Dachsbacher. In: *GPU Pro 4: Advanced Rendering Techniques*. Ed. by Wolfgang Engel. CRC Press, 2013. Chap. Decoupled Deferred Shading on the GPU, pp. 81–97. ISBN: 978-1-4665-6744-3.

Lottes, Timothy. *FXAA*. 2009. URL: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf.

S, Filip. *An investigation of fast real-time GPU-based image blur algorithms*. https://software.intel.com/en-us/blogs/2014/07/15/an-investigation-of-fast-real-time-gpu-based-image-blur-algorithms. Blog. 2014.

Shreiner, Dave et al. In: *OpenGL Programming Guide*. 8th ed. Addison-Wesley, 2013. Chap. Light and Shadow, pp. 368–370. ISBN: 978-0-321-77303-6.

Sousa, Tiago, Nickolay Kasyan, and Nicolas Schulz. In: *GPU Pro 3: Advanced Rendering Techniques*. Ed. by Wolfgang Engel. CRC Press, 2012. Chap. CryENGINE 3: Three Years of Work in Review, p. 160. ISBN: 978-1-4398-8794-3.

Trebilco, Damian. *Light Indexed Deferred Lighting*. 2007. URL: https://github.com/dtrebilco/lightindexed-deferredrender/raw/master/LightIndexedDeferredLighting1.1.pdf.

Glossary of Terms

aliasing artifacts caused when sampling textures and performing calculations, often manifested as rough edges.

API application programming interface.

application programming interface a well-defined standard describing the way in which applications interface with third-party libraries.

bits per pixel number of bits required to represent a single pixel in video memory.

Blinn-Phong reflection model model that approximates the way in which light is reflected off of a surface in the most realistic way.

bloom a blur surrounding edges of very brightly lit objects; usually implemented in conjunction with HDR.

bpp bits per pixel.

compute unit mathematics processing unit which is used to execute a single shader over many data sets in parallel.

deferred shading technique in which all models are rendered into an intermediary buffer, called the geometry buffer, before lighting calculations are applied.

depth test step in the rendering pipeline, where the depth value of a texel is compared to that of a depth buffer to decide whether the texel is drawn or not.

diffuse solid colour of an object, i.e. light that is reflected off of an object, regardless of the viewer's orientation toward it.

fast approximate anti-aliasing a post-processing technique that reduces sharp edges on objects caused by aliasing through edge detection.

float (also **floating point**) system of describing an arbitrary decimal number, rational or irrational, with a high amount of precision; compare to fixed point, where the precision of the fractional and whole components is predetermined.

floating point *See* float.

forward shading technique in which all lighting calculations are performed when a texel is rendered, regardless of whether it shows up in the final output or not.

fps frames per second.

frame rate the rate at which the graphics output of an application (i.e. its frame) is updated.

FXAA fast approximate antialiasing.

geometry buffer a buffer that stores depth, specular highlights, albedo, and surface normals.

GPU Graphics Processing Unit.

graphics accelerator a dedicated piece of hardware (usually in the form of a plug-in card or chip) that is optimized to perform many complex calculations in hardware, in parallel.

graphics pipeline all components of a GPU connected end-to-end, going from input data to a texel on the screen.

HDR high dynamic range.

high dynamic range technique in which all color values are not limited to a finite display range, but are instead represented as floating point values, then normalized into the display range later.

memory bandwidth the maximum continuous rate at which a graphics accelerator can read data from its memory. Usually specified in transfers per second.

multisampling sampling the same object several times, possibly at different coordinates, then producing a single output value.

OpenGL industry standard library used for interfacing with graphics accelerators. Developed by SGI in the 1990s.

pipeline stall occurs when a 'bubble' is allowed to enter the graphics pipeline: either because the GPU must wait on data or calculations, or if a significant state change (such as changing shader programs) takes place.

red green blue the three primary colours used in computer displays to produce any possible colour; also known as additive mixing.

refresh rate the rate at which the video hardware updates the display: the maximum rate at which the screen can be updated.

RGB red, green, blue.

shader code that is executed on the GPU, performing a variety of processing tasks for texels to be output. A GPU may have hundreds of compute units, each of which can run a shader over a given set of data.

shadow mapping a method used to cast shadows, by rendering the scene from the viewpoint of a light source, then transforming the resultant depth when rendering lighting.

skybox static background that is rendered in areas where no objects are drawn to provide a sense of scale and define the environment.

specular bright spots of lights on an object: i.e. the light that is reflected off of an object, based on the angle of the light source relative to the viewer.

supersampling sampling the the same object at a much higher resolution than is needed, in order to produce smoother outputs and combat aliasing artifacts.

texel a single pixel on a texture, usually identified by either a 2D or a 3D point.

tone mapping process of converting HDR colours to display (RGB) colours.

white point a vector that describes where in a colour space white is, in essence defining its origin.

Selected Code Excerpts

Several pieces of relevant program code are included for further reference. They are demarcated by their filename, and their type (shader, `C++` code, etc) is clearly indicated. Some files have additional information, because they may not be referenced elsehwere in the text.

shader/lighting.shader

```
// VERTEX
// A simple vertex shader to handle lighting. It is meant to be rendered onto a
// single quad that fills the entire screen.
#version 400 core
layout (location = 0) in vec3 VtxPosition;
layout (location = 1) in vec2 VtxTexCoord;

out vec2 TexCoord;

void main() {
    gl_Position = vec4(VtxPosition, 1.0f);
    TexCoord = VtxTexCoord;
}

~~~
// FRAGMENT
// All lighting calculation is done in the fragment shader, pulling information
// from the different G buffer components, that were rendered by an earlier
// rendering pass: the position, normals, albedo and specular colours.
#version 400 core
in vec2 TexCoord;

// Output lighted colours.
layout (location = 0) out vec3 FragColour;

// These three textures are rendered into when the geometry is rendered.
uniform sampler2D gDepth;
uniform sampler2D gNormal;
uniform sampler2D gAlbedoSpec;

// Ambient light
struct AmbientLight {
    float Intensity;
    vec3 Colour;
```

```
};

uniform AmbientLight ambientLight;

// Directional, point and spotlight data
struct DirectionalLight {
    // direction pointing FROM light source
    vec3 Direction;

    vec3 DiffuseColour;
    vec3 SpecularColour;
};

const int NUM_DIRECTIONAL_LIGHTS = 4;
uniform DirectionalLight directionalLights[NUM_DIRECTIONAL_LIGHTS];

struct PointLight {
    vec3 Position;

    vec3 DiffuseColour;
    vec3 SpecularColour;
    float Linear;
    float Quadratic;
};

const int NUM_POINT_LIGHTS = 32;
uniform PointLight pointLights[NUM_POINT_LIGHTS];

struct SpotLight {
    vec3 Position;
    vec3 Direction;

    vec3 DiffuseColour;
    vec3 SpecularColour;
    float Linear;
    float Quadratic;

    // cosines of angles
    float InnerCutOff; // when the light begins to fade
    float OuterCutOff; // outside of this angle, no light is produced
};

const int NUM_SPOT_LIGHTS = 8;
uniform SpotLight spotLights[NUM_SPOT_LIGHTS];

// Number of directional, point and spotlights
uniform vec3 LightCount;
// General parameters
uniform vec3 viewPos;

// Inverse projection matrix: view space -> world space
uniform mat4 projMatrixInv;
```

```
// Inverse view matrix: clip space -> view space
uniform mat4 viewMatrixInv;
// Light view matrix: world space -> light space
uniform mat4 lightSpaceMatrix;

// Reconstructs the position from the depth buffer.
vec3 WorldPosFromDepth(float depth);

// 1.0 when x > y, 0.0 otherwise
float when_gt(float x, float y) {
    return max(sign(x - y), 0.0);
}

void main() {
    // Get the depth of the fragment and recalculate the position
    float Depth = texture(gDepth, TexCoord).x;
    vec3 FragWorldPos = WorldPosFromDepth(Depth);

    // retrieve the normals and shininess
    vec4 NormalShiny = texture(gNormal, TexCoord);
    vec3 Normal = NormalShiny.rgb;
    float Shininess = NormalShiny.a;

    // Calculate the view direction
    vec3 viewDir = normalize(viewPos - FragWorldPos);

    // retrieve albedo and specular
    vec4 AlbedoSpec = texture(gAlbedoSpec, TexCoord);
    vec3 Diffuse = AlbedoSpec.rgb;
    float Specular = AlbedoSpec.a;

    // if only the skybox is rendered at a given light, skip lighting
    if(Depth < 1.0) {
        // Ambient lighting
        vec3 ambient = Diffuse * ambientLight.Intensity;
        vec3 lighting = vec3(0, 0, 0);

        // Directional lights
        for(int i = 0; i < LightCount.x;  ++i) {
            // get some info about the light
            DirectionalLight light = directionalLights[i];
            vec3 lightDir = normalize(-light.Direction);

            // Diffuse
            vec3 diffuse = max(dot(Normal, lightDir), 0.0) * Diffuse * light.DiffuseColour;

            // Specular
            vec3 halfwayDir = normalize(lightDir + viewDir);
            float spec = pow(max(dot(Normal, halfwayDir), 0.0), Shininess);

            vec3 specular = spec * Specular * light.SpecularColour;

```

```
            // Output
            lighting += (diffuse + specular);
        }

        // Point lights
        for(int i = 0; i < LightCount.y; ++i) {
            // get some light info
            PointLight light = pointLights[i];
            vec3 lightDir = normalize(light.Position - FragWorldPos);

            // Diffuse
            vec3 diffuse = max(dot(Normal, lightDir), 0.0) * Diffuse * light.DiffuseColour;

            // Specular
            vec3 halfwayDir = normalize(lightDir + viewDir);
            float spec = pow(max(dot(Normal, halfwayDir), 0.0), Shininess);

            vec3 specular = spec * Specular * light.SpecularColour;

            // Attenuation
            float distance = length(light.Position - FragWorldPos);
            float attenuation = 1.0 / (1.0 + (light.Linear * distance) +
                                        (light.Quadratic * distance * distance));

            diffuse *= attenuation;
            specular *= attenuation;

            // Output
            lighting += (diffuse + specular);
        }

        // Spotlights
        for(int i = 0; i < LightCount.z;  ++i) {
            // get some info about the light
            SpotLight light = spotLights[i];

            // Calculate to see whether we're inside the 'cone of influence'
            vec3 lightDir = normalize(light.Position - FragWorldPos);
            float theta = dot(lightDir, normalize(-light.Direction));

            // We're working with cosines, not angles, so >
            if(theta > light.OuterCutOff) {
                // Diffuse
                vec3 diffuse = max(dot(Normal, lightDir), 0.0) * Diffuse * light.DiffuseColour;

                // Specular
                vec3 halfwayDir = normalize(lightDir + viewDir);
                float spec = pow(max(dot(Normal, halfwayDir), 0.0), Shininess);

                vec3 specular = spec * Specular * light.SpecularColour;

                // Spotlight (soft edges)
```

```
                float theta = dot(lightDir, normalize(-light.Direction));
                float epsilon = (light.InnerCutOff - light.OuterCutOff);
                float intensity = clamp((theta - light.OuterCutOff) / epsilon, 0.0, 1.0);
                diffuse *= intensity;
                specular *= intensity;

                // Attenuation
                float distance = length(light.Position - FragWorldPos);
                float attenuation = 1.0f / (1.0 + (light.Linear * distance) +
                                    (light.Quadratic * (distance * distance)));

                diffuse  *= attenuation;
                specular *= attenuation;

                lighting += (diffuse + specular);
            }
        }

        // Combine everything
        lighting += ambient;
        // output colour of the fragment
        FragColour = lighting;
    }
}

// this is supposed to get the world position from the depth buffer
vec3 WorldPosFromDepth(float depth) {
    // Normalize Z
    float ViewZ = (depth * 2.0) - 1.0;
    // Get clip space
    vec4 clipSpacePosition = vec4(TexCoord * 2.0 - 1.0, ViewZ, 1);
    // Clip space -> View space
    vec4 viewSpacePosition = projMatrixInv * clipSpacePosition;
    // Perspective division
    viewSpacePosition /= viewSpacePosition.w;
    // View space -> World space
    vec4 worldSpacePosition = viewMatrixInv * viewSpacePosition;
    // Discard w component
    return worldSpacePosition.xyz;
}
```

shader/model.shader

```
// VERTEX
#version 400 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;

out vec3 WorldPos;
```

```
out vec2 TexCoords;
out vec3 Normal;

uniform mat4 model;
uniform mat4 projectionView; // projection * view
uniform mat3 normalMatrix; // transpose(inverse(mat3(model)))

void main() {
    // Forward the world position and texture coordinates
    vec4 worldPos = model * vec4(position, 1.0f);
    WorldPos = worldPos.xyz;
    TexCoords = texCoords;

    // Set position of the vertex pls
    gl_Position = projectionView * worldPos;

    // Send normals (multiplied by normal matrix)
    Normal = normalMatrix * normal;
}

~~~
// FRAGMENT
#version 400 core

// Material data
struct MaterialStruct {
    // how reflective the material is: lower value = more reflective
    float shininess;
};

// The normal/shininess buffer is colour attachment 0
layout (location = 0) out vec4 gNormal;
// The albedo/specular buffer is colour attachment 1
layout (location = 1) out vec4 gAlbedoSpec;

// Inputs from vertex shader
in vec2 TexCoords;
in vec3 WorldPos;
in vec3 Normal;

// Number of textures to sample (diffuse, specular)
uniform vec2 NumTextures;

// Samplers (for diffuse and specular)
uniform sampler2D texture_diffuse1;
uniform sampler2D texture_diffuse2;
uniform sampler2D texture_diffuse3;
uniform sampler2D texture_diffuse4;

uniform sampler2D texture_specular1;
uniform sampler2D texture_specular2;
uniform sampler2D texture_specular3;
```

```
uniform sampler2D texture_specular4;

// Material data
uniform MaterialStruct Material;

// 1.0 when x == y, 0.0 otherwise
float when_eq(float x, float y) {
    return 1.0 - abs(sign(x - y));
}

// 1.0 when x < y; 0.0 otherwise
float when_lt(float x, float y) {
    return min(1.0 - sign(x - y), 1.0);
}

// 1.0 when x > y; 0.0 otherwise
float when_ge(float x, float y) {
    return 1.0 - when_lt(x, y);
}

void main() {
    // Store the per-fragment normals and material shininess into the gbuffer
    gNormal.rgb = normalize(Normal);
    gNormal.a = Material.shininess;

    // Diffuse per-fragment color
    vec3 diffuse = texture(texture_diffuse1, TexCoords).rgb;

    if(NumTextures.x >= 2) {
        diffuse += texture(texture_diffuse2, TexCoords).rgb;
    } if(NumTextures.x >= 3) {
        diffuse += texture(texture_diffuse3, TexCoords).rgb;
    } if(NumTextures.x >= 4) {
        diffuse += texture(texture_diffuse4, TexCoords).rgb;
    }

    // Specular intensity
    float specular = texture(texture_specular1, TexCoords).a;

    if(NumTextures.y >= 2) {
        specular += texture(texture_specular2, TexCoords).a;
    } if(NumTextures.y >= 3) {
        specular += texture(texture_specular3, TexCoords).a;
    } if(NumTextures.y >= 4) {
        specular += texture(texture_specular4, TexCoords).a;
    }

    // store diffuse colour and specular component
    gAlbedoSpec.rgb = diffuse;
    gAlbedoSpec.a = specular;
}
```

shader/hdr.shader

```
// VERTEX
// A simple vertex shader to handle bloom.
#version 400 core
layout (location = 0) in vec3 VtxPosition;
layout (location = 1) in vec2 VtxTexCoord;

out vec2 TexCoords;

void main() {
    gl_Position = vec4(VtxPosition, 1.0f);
    TexCoords = VtxTexCoord;
}

~~~
// FRAGMENT
#version 400 core

in vec2 TexCoords;

// output the actual colour
layout (location = 0) out vec4 FragColour;

// Raw scene colour output
uniform sampler2D inSceneColours;
// Brightest parts of the scene (pre-blurred, low res)
uniform sampler2D inBloomBlur;

// white point: the brightest colour
uniform vec3 whitePoint;
// exposure value
uniform float exposure;

vec3 TonemapColour(vec3 x);

void main() {
    // additively blend the HDR and bloom textures
    vec3 hdrColour = texture(inSceneColours, TexCoords).rgb;
    vec3 bloomColour = texture(inBloomBlur, TexCoords).rgb;
    hdrColour += bloomColour;

    // apply HDR and white point
    hdrColour = TonemapColour(hdrColour * exposure);
    hdrColour = hdrColour / TonemapColour(whitePoint);

    // compute luma for the FXAA shader
    vec4 outColour = vec4(hdrColour, 0);
    outColour.a = dot(outColour.rgb, vec3(0.299, 0.587, 0.114));

    FragColour = outColour;
```

```
}

// Applies the tonemapping algorithm on a colour
vec3 TonemapColour(vec3 x) {
    float A = 0.15;
    float B = 0.50;
    float C = 0.10;
    float D = 0.20;
    float E = 0.02;
    float F = 0.30;

    return ((x * (A * x + C * B) + D * E) / (x * (A * x + B) + D * F)) - E / F;
}
```

shader/bloom.shader

```
// VERTEX
// A simple vertex shader to handle the blur for bloom.
#version 400 core
layout (location = 0) in vec3 VtxPosition;
layout (location = 1) in vec2 VtxTexCoord;

out vec2 TexCoord;

void main() {
    gl_Position = vec4(VtxPosition, 1.0f);
    TexCoord = VtxTexCoord;
}

~~~
// FRAGMENT
#version 400 core

in vec2 TexCoord;

// output the actual colour
layout (location = 0) out vec4 FragColour;

// The input from the last pass of the shader
uniform sampler2D inTex;

// The direction of the blur: (1, 0) for horizontal, (0, 1) for vertical.
uniform vec2 direction;
// Size of texture
uniform vec2 resolution;

// Performs a 13x13 Gaussian blur.
vec4 blur13(sampler2D image, vec2 uv, vec2 resolution, vec2 direction);

void main() {
```

```
    FragColour = blur13(inTex, TexCoord, resolution, direction);
}

// Performs a 13x13 Gaussian blur.
vec4 blur13(sampler2D image, vec2 uv, vec2 resolution, vec2 direction) {
    vec4 color = vec4(0.0);

    vec2 off1 = vec2(1.411764705882353) * direction;
    vec2 off2 = vec2(3.2941176470588234) * direction;
    vec2 off3 = vec2(5.176470588235294) * direction;

    color += texture(image, uv) * 0.1964825501511404;
    color += texture(image, uv + (off1 / resolution)) * 0.2969069646728344;
    color += texture(image, uv - (off1 / resolution)) * 0.2969069646728344;
    color += texture(image, uv + (off2 / resolution)) * 0.09447039785044732;
    color += texture(image, uv - (off2 / resolution)) * 0.09447039785044732;
    color += texture(image, uv + (off3 / resolution)) * 0.010381362401148057;
    color += texture(image, uv - (off3 / resolution)) * 0.010381362401148057;

    return color;
}
```

render/SceneLighting.cpp: Performs the application of the lighting on a geometry buffer that has previously been rendered into.

```
/*
 * SceneLighting.cpp
 *
 *  Created on: Aug 22, 2015
 *      Author: tristan
 */

#include "SceneLighting.h"
#include "HDRRenderer.h"
#include "FXAARenderer.h"

#include <cassert>
#include <iostream>

#include <glbinding/gl/gl.h>
#include <glbinding/Binding.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include "../level/primitives/lights/DirectionalLight.h"
#include "../level/primitives/lights/PointLight.h"
#include "../level/primitives/lights/SpotLight.h"
```

```
#include "../../housekeeping/ServiceLocator.h"

// vertices for a full-screen quad
static const gl::GLfloat vertices[] = {
    -1.0f,  1.0f, 0.0f,     0.0f, 1.0f,
    -1.0f, -1.0f, 0.0f,     0.0f, 0.0f,
     1.0f,  1.0f, 0.0f,     1.0f, 1.0f,
     1.0f, -1.0f, 0.0f,     1.0f, 0.0f,
};


static const glm::vec3 cubeLightColours[] = {
    glm::vec3(1.0f, 0.0f, 0.0f),
    glm::vec3(0.0f, 1.0f, 0.0f),
    glm::vec3(0.0f, 0.0f, 1.0f),
    glm::vec3(1.0f, 0.5f, 0.0f) * 10.f,
};

static const glm::vec3 cubeLightPositions[] = {
    glm::vec3( 1.5f,  2.0f, -2.5f),
    glm::vec3( 1.5f,  0.2f, -1.5f),
    glm::vec3(-1.3f,  1.0f, -1.5f),
    glm::vec3( 1.5f,  2.0f, -1.5f)
};

#import "skyboxVertices.h"

using namespace gl;
using namespace std;
namespace gfx {

/**
 * Allocates the various textures needed for the G-Buffer.
 */
SceneLighting::SceneLighting() {
    // Load the shader program
    this->program = new ShaderProgram("rsrc/shader/lighting.shader");
    this->program->link();

    // allocate the FBO
    this->fbo = new FrameBuffer();
    this->fbo->bindRW();

    // get size of the viewport
    unsigned int width = ServiceLocator::window()->width;
    unsigned int height = ServiceLocator::window()->height;

    // Normal colour (RGB) and shininess(A) buffer
    this->gNormal = new Texture2D(0);
    this->gNormal->allocateBlank(width, height, Texture2D::RGBA16F);
    this->gNormal->setDebugName("gBufNormal");
```

```

    this->fbo->attachTexture2D(this->gNormal, FrameBuffer::ColourAttachment0);

    // Colour and specular buffer
    this->gAlbedoSpec = new Texture2D(1);
    this->gAlbedoSpec->allocateBlank(width, height, Texture2D::RGBA8);
    this->gAlbedoSpec->setUsesLinearFiltering(true);
    this->gAlbedoSpec->setDebugName("gBufAlbedoSpec");

    this->fbo->attachTexture2D(this->gAlbedoSpec, FrameBuffer::ColourAttachment1);

    // Depth and stencil
    this->gDepth = new Texture2D(2);
    this->gDepth->allocateBlank(width, height, Texture2D::Depth24Stencil8);
    this->gDepth->setDebugName("gBufDepth");

    this->fbo->attachTexture2D(this->gDepth, FrameBuffer::DepthStencil);

    // Specify the buffers used for rendering (sans depth)
    FrameBuffer::AttachmentType buffers[] = {
        FrameBuffer::ColourAttachment0,
        FrameBuffer::ColourAttachment1,
        FrameBuffer::End
    };
    this->fbo->setDrawBuffers(buffers);

    // Ensure completeness of the buffer.
    assert(FrameBuffer::isComplete() == true);
    FrameBuffer::unbindRW();

    // set up a VAO and VBO for the full-screen quad
    vao = new VertexArray();
    vbo = new Buffer(Buffer::Array, Buffer::StaticDraw);

    vao->bind();
    vbo->bind();

    vbo->bufferData(sizeof(vertices), (void *) vertices);

    vao->registerVertexAttribPointer(0, 3, VertexArray::Float,
                                     5 * sizeof(GLfloat), 0);
    vao->registerVertexAttribPointer(1, 2, VertexArray::Float,
                                     5 * sizeof(GLfloat), 3 * sizeof(GLfloat));

    VertexArray::unbind();

    // tell our program which texture units are used
    this->program->bind();
    this->program->setUniform1i("gNormal", this->gNormal->unit);
    this->program->setUniform1i("gAlbedoSpec", this->gAlbedoSpec->unit);
    this->program->setUniform1i("gDepth", this->gDepth->unit);

```

```
    // Compile skybox shader and set up some vertex data
    this->skyboxProgram = new ShaderProgram("rsrc/shader/skybox.shader");
    this->skyboxProgram->link();

    vaoSkybox = new VertexArray();
    vboSkybox = new Buffer(Buffer::Array);

    vaoSkybox->bind();
    vboSkybox->bind();

    vboSkybox->bufferData(sizeof(skyboxVertices), (void *) skyboxVertices);

    vaoSkybox->registerVertexAttribPointer(0, 3, VertexArray::Float,
                                           3 * sizeof(GLfloat), 0);
    VertexArray::unbind();

    // load cubemap texture
    this->skyboxTexture = new TextureCube(0);
    this->skyboxTexture->setDebugName("SkyCube");
    this->skyboxTexture->loadFromImages("rsrc/tex/cube/", true);
    TextureCube::unbind();

    // set up test lights
    this->setUpTestLights();
}

/**
 * Sets up the default lights for testing.
 */
void SceneLighting::setUpTestLights(void) {
    // set up a test directional light
    DirectionalLight *dir = new DirectionalLight();
    dir->setDirection(glm::vec3(-0.2f, -1.0f, -0.3f));
    dir->setColour(glm::vec3(0.85, 0.85, 0.75));

    this->addLight(dir);

    // set up a test spot light
    this->spot = new SpotLight();
    this->spot->setInnerCutOff(12.5f);
    this->spot->setOuterCutOff(17.5f);
    this->spot->setLinearAttenuation(0.1f);
    this->spot->setQuadraticAttenuation(0.8f);
    this->spot->setColour(glm::vec3(1.0f, 0.33f, 0.33f));

    this->addLight(this->spot);

    // point lights
    for(int i = 0; i < 4; i++) {
        PointLight *light = new PointLight();

        light->setPosition(glm::vec3(cubeLightPositions[i]));
```

```
        light->setColour(glm::vec3(cubeLightColours[i]));

        light->setLinearAttenuation(0.7f);
        light->setQuadraticAttenuation(1.8f);

        this->addLight(light);
    }
}

SceneLighting::~SceneLighting() {
    delete this->program;
    delete this->fbo;
    delete this->gAlbedoSpec;
    delete this->gDepth;
    delete this->gNormal;
    delete this->vao;
    delete this->vbo;

    // delete the lights
    for(auto &light : this->lights) {
        delete light;
    }
}

/**
 * Clear the output buffer
 */
void SceneLighting::beforeRender(void) {
    // clear the output buffer
    glClear(GL_COLOR_BUFFER_BIT);

    // ensure we do not write to the depth buffer during lighting
    glDepthMask(GL_FALSE);
}

/**
 * Renders the lighting pass.
 */
void SceneLighting::render(void) {
    // use our lighting shader, bind textures and set their locations
    this->program->bind();

    this->gNormal->bind();
    this->gAlbedoSpec->bind();
    this->gDepth->bind();

    if(this->shadowTexture) {
        this->shadowTexture->bind();
    }

    // Send ambient light
    this->program->setUniform1f("ambientLight.Intensity", 0.05f);
```

```
    this->program->setUniformVec("ambientLight.Colour", glm::vec3(1.0, 1.0, 1.0));

    // send the different types of light
    this->spot->setDirection(this->viewDirection);
    this->spot->setPosition(this->viewPosition);

    this->sendLightsToShader();

    // send the camera position and inverse view matrix
    this->program->setUniformVec("viewPos", this->viewPosition);

    // Inverse projection and view matrix
    glm::mat4 viewMatrixInv = glm::inverse(this->viewMatrix);
    this->program->setUniformMatrix("viewMatrixInv", viewMatrixInv);

    glm::mat4 projMatrixInv = glm::inverse(this->projectionMatrix);
    this->program->setUniformMatrix("projMatrixInv", projMatrixInv);

    // render a full-screen quad
    vao->bind();
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    VertexArray::unbind();

    // unbind textures
    this->gNormal->unbind();
    this->gAlbedoSpec->unbind();
    this->gDepth->unbind();

    // render the skybox
    renderSkybox();
}

/**
 * Sends the different lights' data to the shader, which is currently bound.
 */
void SceneLighting::sendLightsToShader(void) {
    // set up counters
    int numDirectional, numPoint, numSpot;
    numDirectional = numPoint = numSpot = 0;

    // go through each type of light
    for(auto &light : this->lights) {
        switch(light->getType()) {
            case lights::AbstractLight::Directional:
                light->sendToProgram(numDirectional++, this->program);
                break;

            case lights::AbstractLight::Point:
                light->sendToProgram(numPoint++, this->program);
                break;

            case lights::AbstractLight::Spot:
```

```
                 light->sendToProgram(numSpot++, this->program);
                break;

            default:
                cerr << "Unknown light type: " << light->getType() << endl;
                break;
        }
    }

    // send how many of each type of light (directional, point, spot) we have
    glm::vec3 lightNums = glm::vec3(numDirectional, numPoint, numSpot);
    this->program->setUniformVec("LightCount", lightNums);
}

/**
 * Renders the skybox.
 */
void SceneLighting::renderSkybox(void) {
    // set up some state for the skybox
    glDepthFunc(GL_LEQUAL);

    this->skyboxProgram->bind();

    // calculate a new view matrix with translation components removed
    glm::mat4 newView = glm::mat4(glm::mat3(this->viewMatrix));

    this->skyboxProgram->setUniformMatrix("view", newView);
    this->skyboxProgram->setUniformMatrix("projection", this->projectionMatrix);

    // bind VAO, texture, then draw
    this->vaoSkybox->bind();

    this->skyboxTexture->bind();
    this->skyboxProgram->setUniform1i("skyboxTex", this->skyboxTexture->unit);

    glDrawArrays(GL_TRIANGLES, 0, 36);
}

/**
 * Unbinds any information and prepares the next frame.
 */
void SceneLighting::afterRender(void) {
    // allow successive render passes to render depth
    glDepthMask(GL_TRUE);
}

/**
 * Binds the various G-buffer elements before the scene itself is rendered. This
 * sets up three textures, into which the following data is rendered:
 *
 * 1. Positions (RGB)
 * 2. Colour (RGB) plus specular (A)
```

```
 * 3. Normal vectors (RGB)
 *
 * Following a call to this function, the scene should be rendered, and when
 * this technique is rendered, it will render the final geometry with lighting
 * applied.
 */
void SceneLighting::bindGBuffer(void) {
    this->fbo->bindRW();

    // re-attach the depth texture
    this->fbo->attachTexture2D(this->gDepth, FrameBuffer::DepthStencil);
    assert(FrameBuffer::isComplete() == true);
}

/**
 * Sets the HDR renderer's framebuffer to use our depth stencil.
 */
void SceneLighting::setHDRRenderer(HDRRenderer *renderer) {
    renderer->setDepthBuffer(this->gDepth, true);
}

/**
 * Sets the FXAA renderer to re-use our albedo texture.
 */
void SceneLighting::setFXAARenderer(FXAARenderer *renderer) {
    renderer->setColourInputTex(this->gAlbedoSpec);
}

/**
 * Adds a light to the list of lights. Each frame, these lights are sent to the
 * GPU.
 *
 * @note We assume ownership of the objects once they are added, so they are
 * deleted when this class goes away.
 */
void SceneLighting::addLight(lights::AbstractLight *light) {
    this->lights.push_back(light);
}

/**
 * Removes a previously added light.
 *
 * @return 0 if the light was removed, -1 otherwise.
 */
int SceneLighting::removeLight(lights::AbstractLight *light) {
    auto position = std::find(this->lights.begin(),
                              this->lights.end(), light);

    if(position > this->lights.end()) {
        return -1;
    } else {
        this->lights.erase(position);
```

```
        return 0;
    }
}

/**
 * Sets the texture in which the shadow data is stored, as well as the light
 * space matrix.
 */
void SceneLighting::setShadowTexture(Texture2D *tex, glm::mat4 lightSpaceMtx) {
    // set texture, if it changed
    if(this->shadowTexture != tex) {
        this->shadowTexture = tex;
        this->shadowLightSpaceTransform = lightSpaceMtx;
    }

    // bind program and send texture param
    this->program->bind();
    this->program->setUniform1i("gShadowMap", this->shadowTexture->unit);

    // send the light space matrix's inverse
    glm::mat4 matrix = glm::inverse(lightSpaceMtx);
    this->program->setUniformMatrix("lightToViewMtx", matrix);
}
}
```

render/HDRRenderer.cpp: Applies the blooming blur, gamma correction, and HDR tone mapping.

```cpp
/*
 * HDRRenderer.cpp
 *
 *  Created on: Aug 26, 2015
 *      Author: tristan
 */

#include "HDRRenderer.h"

#include <cassert>
#include <iostream>

#include <glbinding/gl/gl.h>
#include <glbinding/Binding.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include "../../housekeeping/ServiceLocator.h"

// vertices for a full-screen quad
static const gl::GLfloat vertices[] = {
    -1.0f,  1.0f, 0.0f,     0.0f, 1.0f,
    -1.0f, -1.0f, 0.0f,     0.0f, 0.0f,
     1.0f,  1.0f, 0.0f,     1.0f, 1.0f,
     1.0f, -1.0f, 0.0f,     1.0f, 0.0f,
};

using namespace gl;
using namespace std;
namespace gfx {
/**
 * Sets up a basic framebuffer with a floating-point colour attachment.
 */
HDRRenderer::HDRRenderer() {
    // Load the shader program
    this->program = new ShaderProgram("rsrc/shader/hdr.shader");
    this->program->link();

    // set up the framebuffers
    setUpInputBuffers();

    // set up a VAO and VBO for the full-screen quad
    vao = new VertexArray();
    vbo = new Buffer(Buffer::Array, Buffer::StaticDraw);

```

```
    vao->bind();
    vbo->bind();

    vbo->bufferData(sizeof(vertices), (void *) vertices);

    vao->registerVertexAttribPointer(0, 3, VertexArray::Float,
                                     5 * sizeof(GLfloat), 0);
    vao->registerVertexAttribPointer(1, 2, VertexArray::Float,
                                     5 * sizeof(GLfloat), 3 * sizeof(GLfloat));

    VertexArray::unbind();
}

/**
 * Sets up the framebuffer into which the previous rendering stage will output.
 */
void HDRRenderer::setUpInputBuffers(void) {
    // allocate the FBO
    this->inFBO = new FrameBuffer();
    this->inFBO->bindRW();

    // get size of the viewport
    unsigned int width = ServiceLocator::window()->width;
    unsigned int height = ServiceLocator::window()->height;

    // colour (RGB) buffer (gets the full range of lighting values from scene)
    this->inColour = new Texture2D(1);
    this->inColour->allocateBlank(width, height, Texture2D::RGB16F);
    this->inColour->setDebugName("HDRColourIn");

    this->inFBO->attachTexture2D(this->inColour, FrameBuffer::ColourAttachment0);

    // Specify the buffers used for rendering
    FrameBuffer::AttachmentType buffers[] = {
        FrameBuffer::ColourAttachment0,
        FrameBuffer::End
    };
    this->inFBO->setDrawBuffers(buffers);

    // Ensure completeness of the buffer.
    assert(FrameBuffer::isComplete() == true);
    FrameBuffer::unbindRW();

    // tell our program which texture units are used
    this->program->bind();
    this->program->setUniform1i("texInColour", this->inColour->unit);
}

HDRRenderer::~HDRRenderer() {
    delete this->program;
    delete this->inFBO;
    delete this->inColour;
```

```
    delete this->vao;
    delete this->vbo;
}

/**
 * Sets up for HDR rendering.
 */
void HDRRenderer::beforeRender(void) {
    // bind to the window framebuffer
    glDisable(GL_DEPTH_TEST);
}

/**
 * Extracts all the extra bright colours from the render buffer, and forwards
 * them to a different buffer.
 */
void HDRRenderer::render(void) {
    // use the "HDR" shader to get the bright areas to a separate buffer
    this->program->bind();
    this->inColour->bind();

    // render a full-screen quad
    vao->bind();
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
}

/**
 * Binds the HDR buffer.
 */
void HDRRenderer::bindHDRBuffer(void) {
    this->inFBO->bindRW();
}

/**
 * Sets the depth buffer that's attached to the FBO.
 */
void HDRRenderer::setDepthBuffer(Texture2D *depth, bool hasStencil) {
    // check if the texture changed
    if(this->inDepth == depth) { return; } else {
        if(this->inDepth != NULL) {
            this->inFBO->attachTexture2D(this->inDepth, FrameBuffer::DepthStencil);
            assert(FrameBuffer::isComplete() == true);
            return;
        }
    }
    this->inDepth = depth;

    // attach the texture
    this->inFBO->bindRW();
    this->inFBO->attachTexture2D(this->inDepth, FrameBuffer::DepthStencil);

    assert(FrameBuffer::isComplete() == true);
```

```
    FrameBuffer::unbindRW();
}

/**
 * Sets the bloom renderer's framebuffer to use the colour texture.
 */
void HDRRenderer::setBloomRenderer(BloomRenderer *renderer) {
    renderer->setColourInputTex(this->inColour);
}
}
```