Deferred Rendering in Leadwerks Engine

Josh Klint
CEO, Leadwerks Corporation

1.1 Introduction

Deferred lighting is a relatively new rendering concept that performs lighting as a kind of post-processing step. The immediate advantage of this is that lighting is decoupled from scene complexity. Lighting is processed at the same speed whether one or 1,000,000 polygons are onscreen.

Whereas a forward renderer will usually draw to a color and depth buffer, deferred shading uses an additional screen space normal buffer. The combination of normal and depth data is sufficient to calculate lighting in a post-processing step, which is then combined with the color buffer.

Figure 1.0 Deferred lighting performance is independent from scene complexity.

1.2 Forward Rendering in Leadwerks Engine 2.0

Leadwerks Engine 2.0 utilized a forward renderer**.** In order to optimize the engine to render only the visible lights onscreen, we implemented a concept we call "shader groups". Instead of storing a single shader per material, the engine stores an array of versions of that shader, each auto-generated using GLSL defines to control the number of lights and other settings. This approach suffered from several problems that became apparent once we started using the engine in a production environment:

- Each shader group had over 100 potential versions. The constant shader compiling during runtime when entering a new area caused unacceptable pauses. Although

only a few variations ever got compiled and used, it was impossible to guess which ones would be needed. Because there were so many potential variations it was impossible to pre-compile all of these.
- The length of the shaders became quite long, causing slower compiling times and risked exceeding the instruction limit on older hardware.
- Because materials used shader groups and not single shaders, it was difficult or impossible for our end users to alter a shader uniform for material shaders.
- It was difficult to control per-light settings. A single global binary setting would double the number of potential shaders in a group. A per-light binary setting would multiply the shader count by 16 (two settings times a maximum of eight lights).
- The maximum number of lights visible at any given time was capped at eight due to hardware limitations and practicality.

These issues motivated us to research a deferred approach for version 2.1 of our engine.

1.3 Deferred Rendering in Leadwerks Engine 2.1

We set out to implement a deferred renderer with a few conditions in mind. We wanted to create a minimal frame buffer format to minimize bandwidth costs, while supporting all the lighting features of our forward renderer. Additionally, we chose to perform lighting calculations in screen space rather than the world space methods used by the forward renderer. Although we did not know how performance would compare, it was certain from the beginning that deferred lighting would offer a simpler renderer that was easier to control. Additionally, the following advantages applied:

- Easy control of per-light settings. Does this light cast shadows? Does it create a specular reflection?
- Smaller shader sources and faster compiling. There are 32 potential light shaders that still get compiled on-the-fly, but the pauses this causes are much less frequent and barely perceptible.
- No limit to the number of lights rendered.
- Easy addition of new light types.

Because deferred lighting only performs lighting on the final screen pixels, the same characteristic that makes deferred lighting so desirable creates a few new problems:

- Alpha blending becomes difficult.
- No MSAA hardware anti-aliasing.

We found that a modulate (darken) blend can be used for most transparencies, without disrupting the lighting environment. Alpha blended materials can be drawn in a second pass, as we already do for refractive surfaces. A sort of pseudo anti-alias can be performed with a blur post-filter weighted with an edge detection filter.

1.31 The Frame Buffer Format

It is often suggested that a deferred renderer should use a 128-bit GL_RGBA32F float buffer for recording fragment positions. We saw no need for transferring this extra data. Instead, the light shaders reconstruct the fragment screen space position from the screen coordinate and depth value. We found that a 24-bit depth buffer provided good precision for use with shadow map lookups.

The fragment coordinate was used to calculate the screen space position. Here, the *buffersize* uniform is used to pass the screen width and height to the shader:

```
vec3 screencoord;
screencoord = vec3(((gl_FragCoord.x/buffersize.x)-0.5) * 2.0,((-gl_FragCoord.y/buffersize.y)+0.5) * 2.0 /
(buffersize.x/buffersize.y),DepthToZPosition( depth ));
screencoord.x *= screencoord.z;
screencoord.y *= -screencoord.z;
```

The depth was converted to a screen space z position with the following function. The *camerarange* uniform stores the camera near and far clipping distances:

```
float DepthToZPosition(in float depth) {
        return camerarange.x / (camerarange.y - depth * (camerarange.y - camerarange.x)) *
camerarange.y;
}
```

This approach allowed us to save about half the bandwidth costs a frame buffer format with a 128-bit position buffer would have required.

We found that for diffuse lighting, a 32-bit GL_RGBA8 normal buffer yielded good results. However, surfaces with a high specular factor exhibited banding caused by the limited resolution. We tried the GL_RGB10_A2 format, which was precise enough for the normal, but lacked sufficient precision to store the specular value in the alpha channel. We settled on using a 64-bit GL_RGBA16F buffer on supported hardware, with a fallback for a 32-bit GL_RGBA8 buffer.

Specular intensity was packed into the unused alpha channel of the normal buffer. Since graphics hardware prefers data in 32-bit packets, it is likely that the RGBA format is the same internally as the RGB format on most hardware, so this extra data transfer came at no cost. The size of the frame buffer format displayed here is either 120 or 88 bits per pixel, depending on the normal buffer format.

Figure 1.1 Frame buffer format:

Buffer	Format	Bits	Values			
color	GL_RGBA8	32	red	green	blue	alpha
depth	GL_DEPTH_COMPONENT24	24	depth			
normal	GL_RGBA16F or GL_RGBA8	64 or 32	x	y	z	specular factor

Since we were already using multiple render targets, we decided to add support for attaching additional color buffers. This allowed the end user to create an additional buffer for rendering selective bloom or other effects, and allows for extended rendering features without altering the engine source code. The light source for the orange point light pictured here is a good candidate for selective bloom, as our full-bright and lens flare effects did not translate well to the deferred renderer.

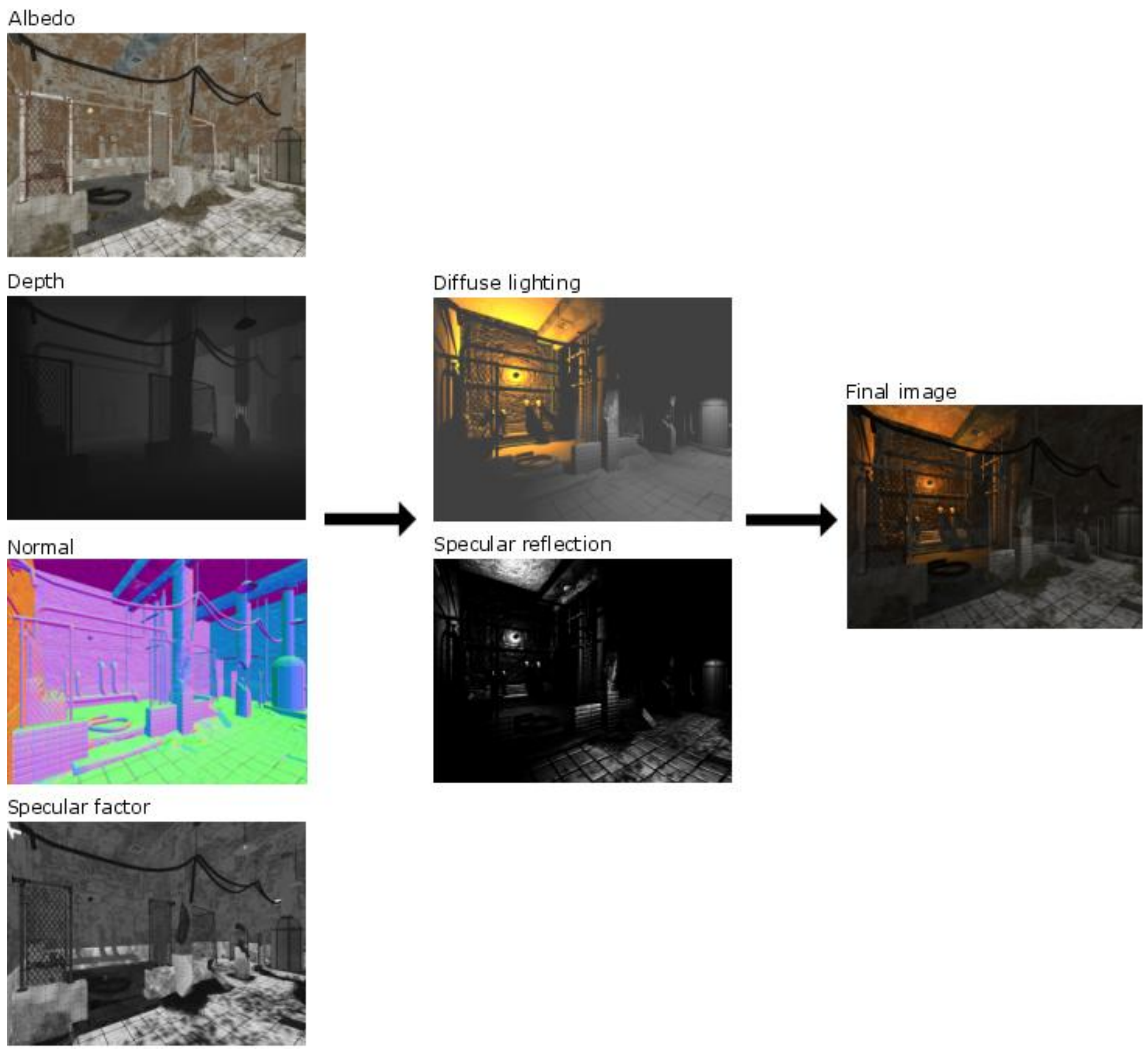

Figure 1.2 Depth, normal, and specular factor are used to calculate diffuse lighting and specular reflection, which is then combined with the albedo color for the final image.

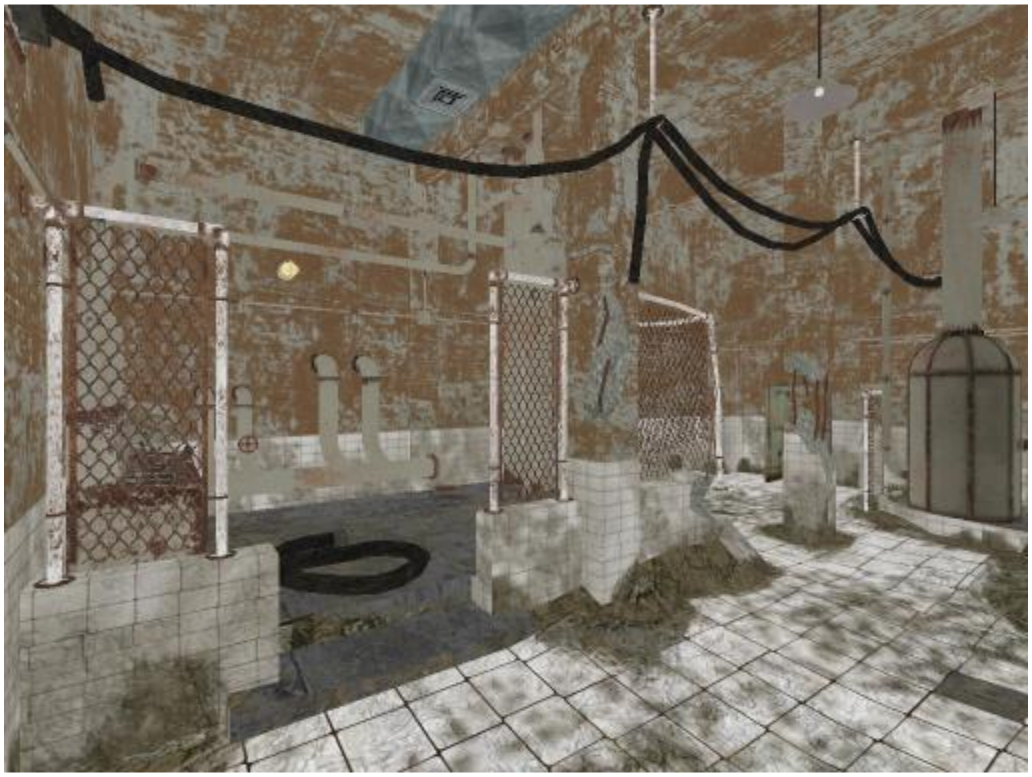

Figure 1.3 Albedo

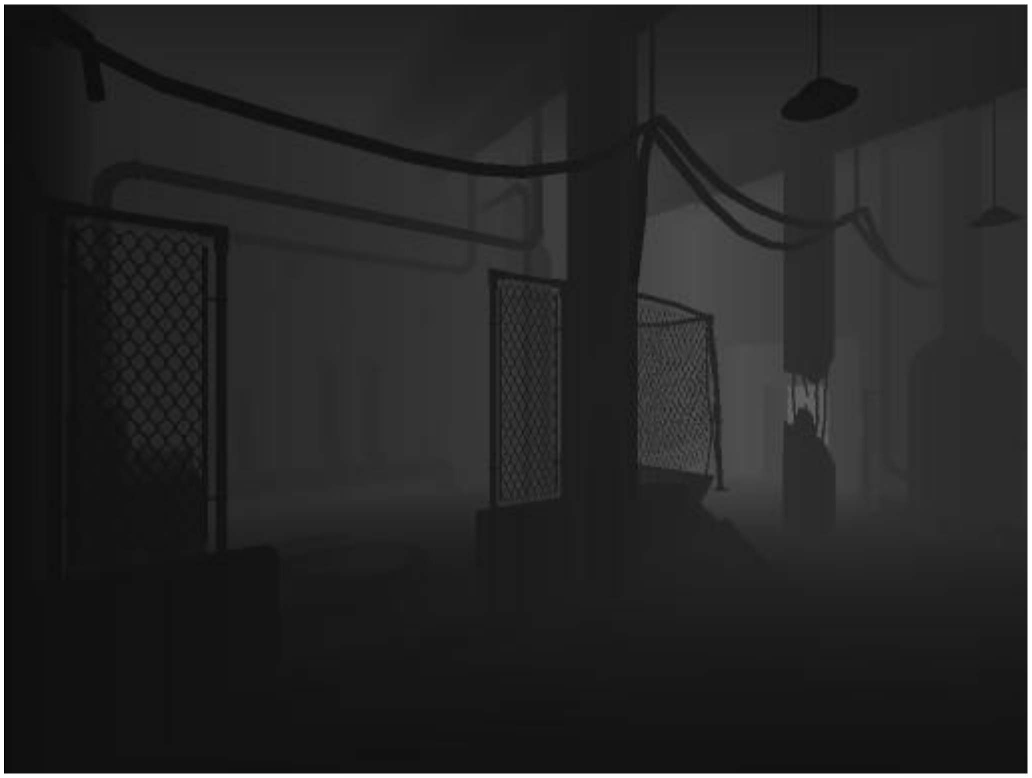

Figure 1.4 Depth

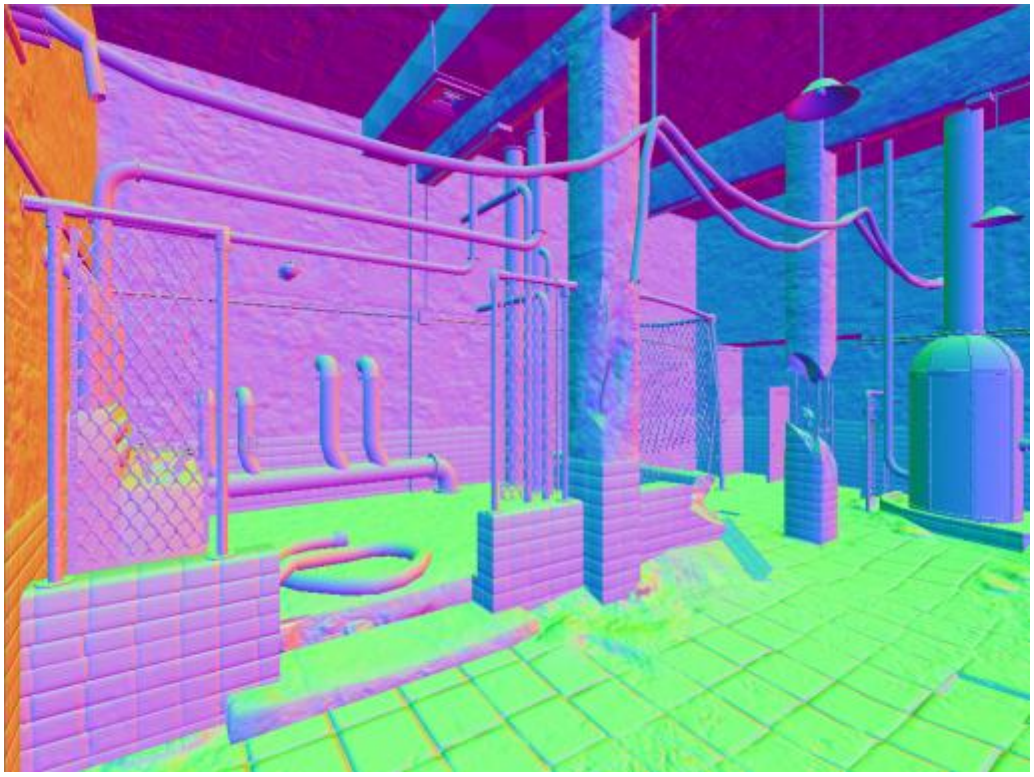

Figure 1.5 Normal

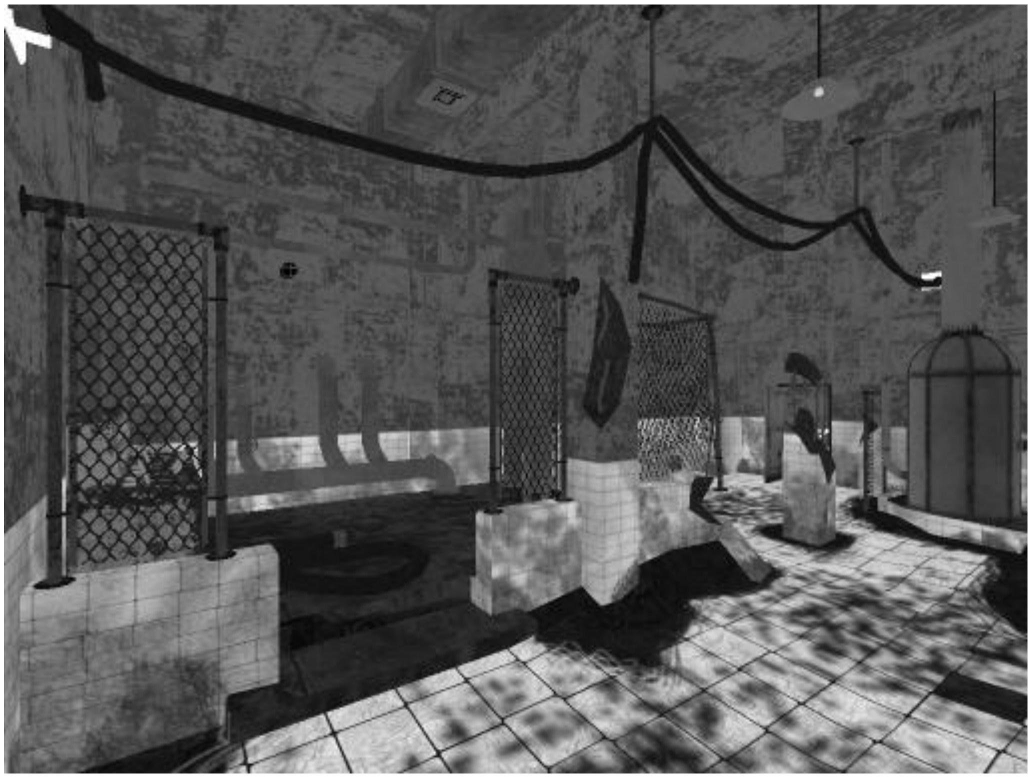

Figure 1.6 Specular factor

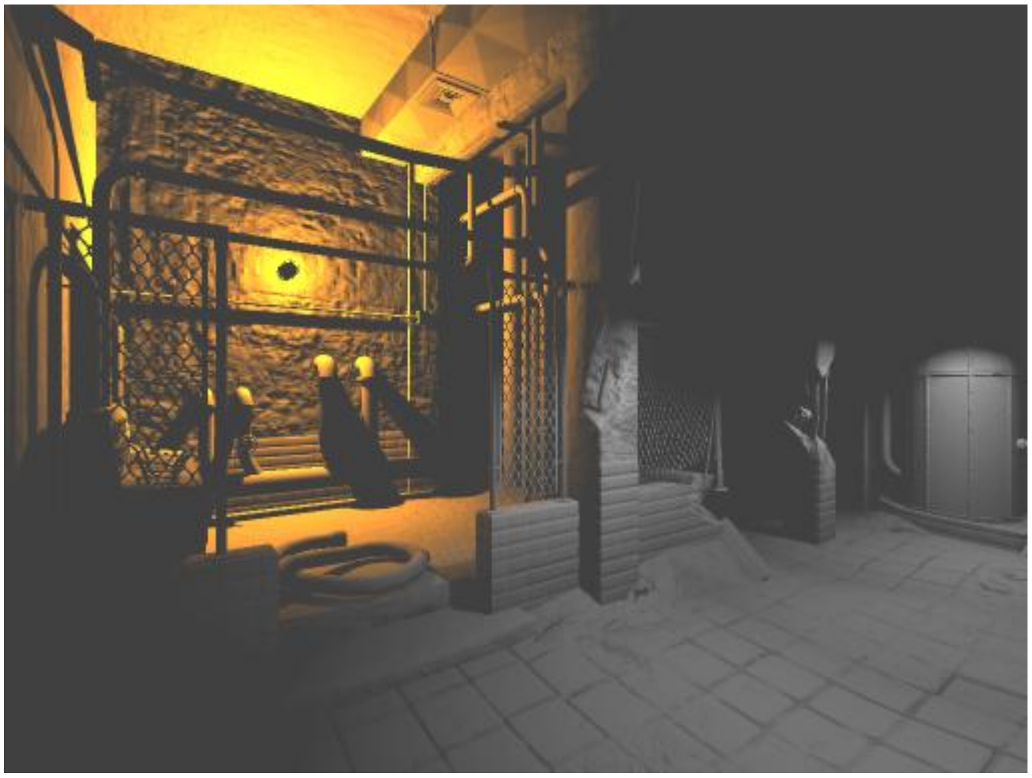

Figure 1.7 Diffuse lighting

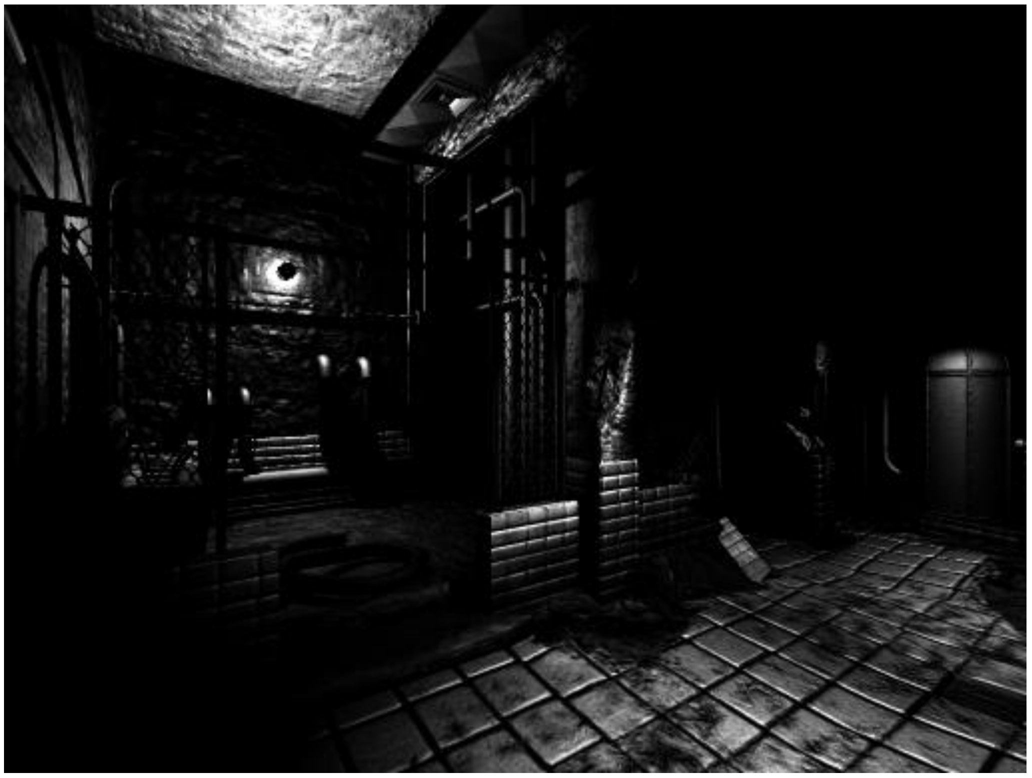

Figure 1.8 Specular reflection

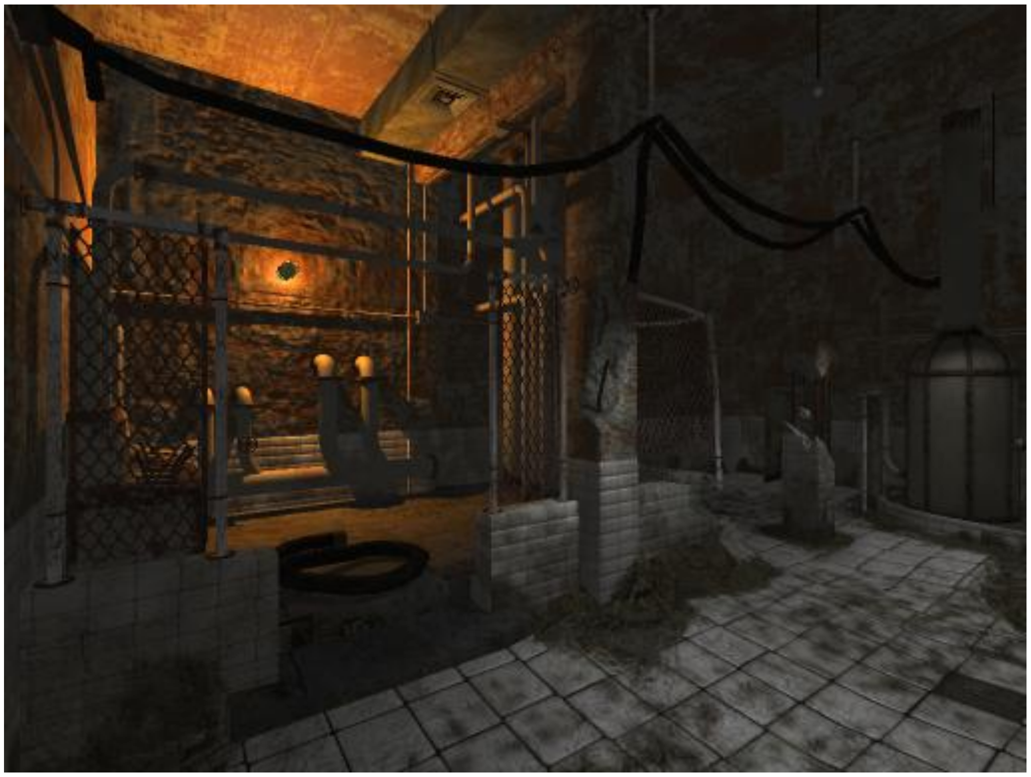

Figure 1.9 Final image

1.32 Optimization

Our engine performs hierarchal culling based on the scene graph, followed by per-object frustum culling. Hardware occlusion queries are performed on the remaining objects. This ensures that only visible lights are updated and drawn. For the remaining visible lights, it is important to process only the fragments that are actually affected by the light source.

The most efficient way to eliminate fragments is to discard them before they are run through the fragment program. We copied the depth value from the depth buffer to the next render target during the first full-screen pass, which is either the ambient pass or the combined ambient and first directional light pass. Point and spot light volumes were then rendered with depth testing enabled. An intersection test detected whether the camera was inside or outside the light volume, and the polygon order and depth function were switched accordingly. Although writing to the depth buffer may disable hierarchal depth culling, an overall performance improvement was observed with depth writing and testing enabled. Further research using the stencil buffer to discard fragments may prove beneficial.

1.4 Results

Our results disprove some common assumptions about deferred rendering. It is often suggested that deferred rendering is only advantageous in scenes with a high degree of overdraw. In every test run we found our deferred renderer to be at least 20% faster than forward lighting. Due to our compact frame buffer format, the increased bandwidth cost was only about four frames per second lost on ATI Radeon HD 3870 and NVidia GEForce 8800 graphics cards. On average we saw a performance improvement of about 50% per light onscreen. A scene with eight visible lights ran four times faster in our deferred renderer.

The speed improvements of our deferred renderer were not limited to high-end hardware. We saw similar performance gains on ATI Radeon X1550 and NVidia GEForce 7200 graphics cards.

Except for the aforementioned issues with MSAA and alpha blending, the appearance of our deferred renderer was identical to that of our forward renderer. Our lens flare effect was abandoned, since it interfered with lighting and could be replaced with more advanced depth-based lighting effects.

1.5 Conclusion

It is inevitable that a fundamental shift in rendering paradigms will cause some incompatibility with previously utilized techniques. However, the simplicity and speed of deferred rendering far outweigh the few disadvantages. Not only did deferred lighting simplify our renderer and eliminate the problems our forward renderer exhibited, but it also yielded a significant performance improvement, even in situations where it is usually suggested a forward renderer would be faster.

1.6 References

Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. 1990. *Computer Graphics: Principles and Practice*. Addison-Wesley.

Shishkovtso, Oles 2005. “GPU Gems 2. Chaper 9 - Deferred Rendering in S.T.A.L.K.E.R.” http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html

Valient, Michael. 2007. "Deferred Rendering in Killzone 2." Presentation at Develop Conference in Brighton, July 2007. http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf

1.7 Acknowledgements

Thanks to Arne Reiners, Rich DiGiovanni, and Emil Persson.