Real Time Rendering Engine

Senior Project – June 2013

Kevin Ubay-Ubay
California Polytechnic State University
San Luis Obispo
Winter and Spring 2013
Advisor: Professor Zoe J. Wood

Table of Contents

Introduction

Entertaining and playable content in computer graphics requires real time rendering. Real time rendering essentially demands that frames need to be rendered within milliseconds in order to deliver an interactive experience for the client. Video games are an example of such a medium that needs real time rendering. Without frame rates in the realm of real time, video games cannot deliver an enjoyable experience. Behind virtually every video game is an engine. The architecture of video game engines usually comprises of sub engines that are specifically designed to handle physics, audio, user input and rendering. The purpose of this project is to focus on one aspect of a video game engine, which is real time rendering.

Various rendering techniques and algorithms were investigated and implemented. Visual quality is an important factor. However, quality often comes at the cost of speed and performance. Compromises in quality and performance have to be made. Ongoing research in various rendering algorithms and techniques search for faster ways to do things while increasing quality.

Related Work

Projects such as OGRE 3D (Object-Oriented Graphics Rendering Engine - http://www.ogre3d.org/) offer an open source rendering engine that developers can utilize to handle the rendering in their video game engines. OGRE 3D handles tasks expected of a rendering engine such as the loading of meshes, spatial querying, scene organization/management and animation (skeletal).

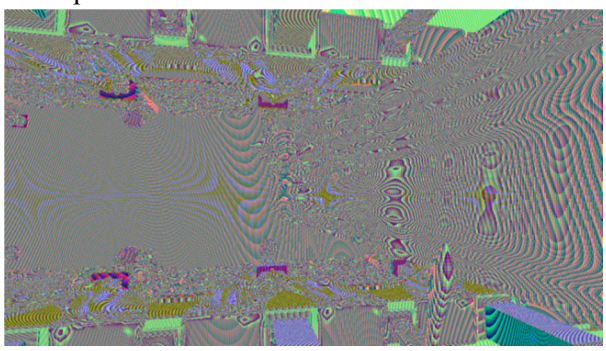
Well featured commercial solutions also exist such as Crytek's *CryEngine* and Epic Game's *Unreal Engine.* In particular, CryEngine employs many rendering techniques to achieve its visual quality. CryEngine uses Light Propagation Volumes as its main global illumination algorithm allowing light bounces and color bleeding done in real time with no pre-computation necessary. Other advanced techniques implemented are real-time local reflections (RLR) which are reflections ray traced within screen space, deferred lighting, light shafts, HDR lighting, normal maps and offset bump mapping. One algorithm in particular used in CryEngine is Screen Space Directional Occlusion which is an improvement over the old screen space ambient occlusion technique (SSAO). This algorithm will be implemented and discussed in this project.

Algorithms & Techniques

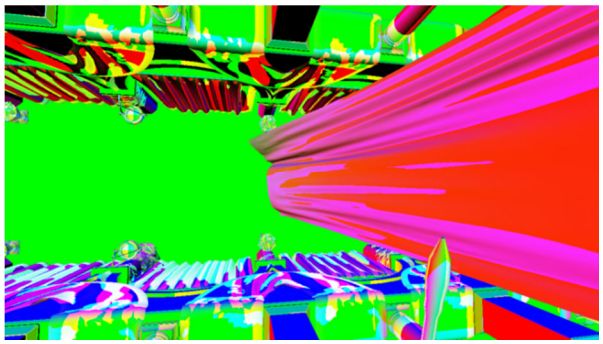
Deferred Shading

Implemented in the project is a deferred rendering pipeline used to decouple the geometry from the lighting. Positions, normal and diffuse are stored as textures in a structure called the G-Buffer. While a G-Buffer requires additional memory for storage, the advantage of the G-Buffer is the information available and stored which can be later used for post processing. For example, screen space techniques such as SSDO and FXAA will make use of the G-Buffer. Regarding lighting, the final lighting is computed in a single pass. This is advantageous in comparison to forward rendering as lighting does not have to be computed in multiple passes (for each rendered object). As such, multiple (as in many) light sources can be supported allowing more detailed lighting in the scenery. Quadratic light attenuation for light sources have also been implemented to prevent over saturation.

Example G-Buffer:

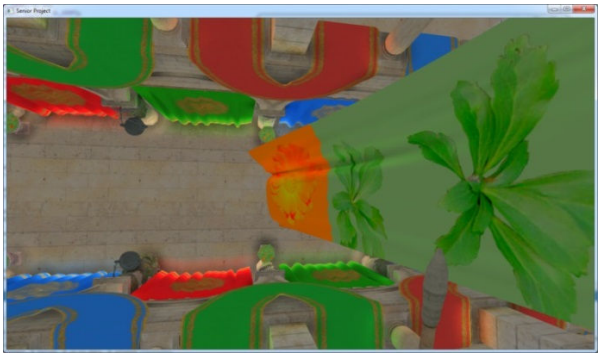


Position



Normal

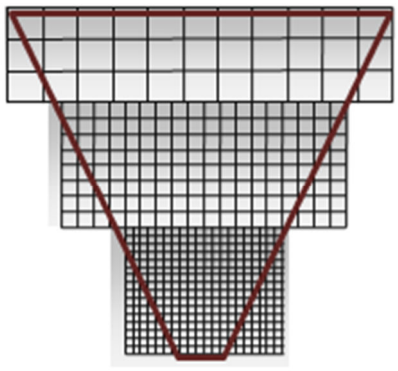
Diffuse



Final

Cascaded Shadow Mapping

Cascaded shadow mapping is a popular technique that splits a shadow map into cascades along a view frustum to minimize perspective aliasing to increase quality as such in the diagram below:



As seen in the diagram, the shadows maps for each cascade are used more efficiently as they are fitted to the view frustum. This technique allows for more detailed shadow maps for objects closer to the viewer as each texel now covers a smaller area of the view frustum (at least for the first cascade near to the viewer's eye) thereby increasing shadow accuracy.

The algorithm follows this flow:

1. Each light source has its own frustum divided into n number of cascades. Render the depth from the light's point of view.
2. Render the scene from the camera's point of view. Get the z depth value.
3. Determine which cascade the z depth value falls into. Use that cascade's shadow map.

Typically, the number of cascade splits used is 4. In implementation, for each light source in the scene, N number (N = number of cascades) of depth textures (shadow maps) are allocated. The next step is to determine where to split (Z-values) along the frustum in camera eye space. NVidia's paper describing the implementation on cascaded shadow maps suggests that the splits should be distributed exponentially by using the formula [2]:

$$z_i = \lambda n(f/n)^{i/N} + (1-\lambda)(n + (i/N)(f-n))$$

Where n and f are the camera's near and far values respectively, N is the total number of splits and i indicates the current split to calculate. λ is a correction factor needed for low numbers of splits (such as 4) that is chosen by the user. Once the splits have been computed, the corners of the frustum planes in camera space for each split are computed via this routine (split points are fed into float variable camZ):

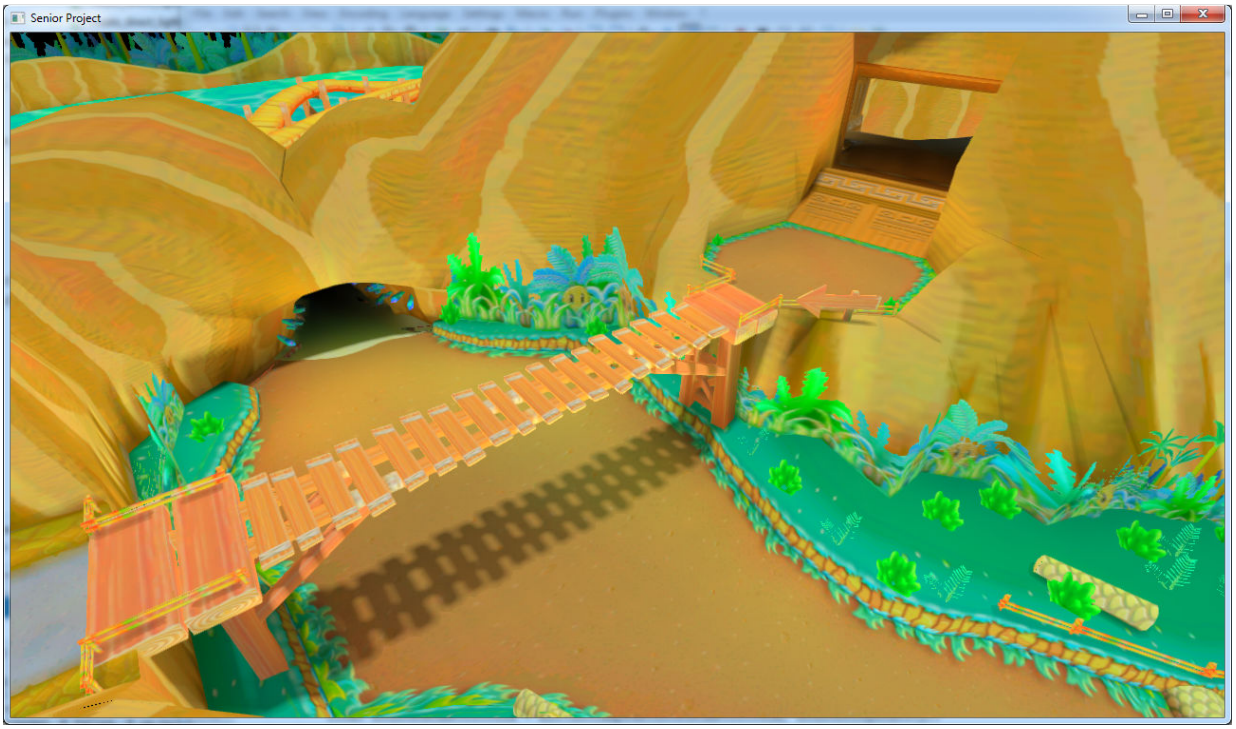
```
void getFrustumPlane(
	const glm::vec3 camPos,
	const glm::vec3 camDir,
	const glm::vec3 camUp,
	float camRatio,
	float camFov,
	float camZ,
	glm::vec3& corner0,
	glm::vec3& corner1,
	glm::vec3& corner2,
	glm::vec3& corner3)
{
	float height = tan(camFov * 0.5f) * camZ;
	float hidth = height * camRatio;
	glm::vec3 center = camPos + camDir * camZ;
	glm::vec3 right = glm::normalize(glm::cross(camDir, camUp));

	corner0 = center - (camUp * height) - (right * width);
	corner1 = center - (camUp * height) + (right * width);
	corner2 = center + (camUp * height) + (right * width);
	corner3 = center + (camUp * height) - (right * width);
}
```

With view matrix is constructed in the light's direction, each corner of the frustum plane is transformed into the light's space. The transformed plane points are then used to generate an axis aligned bounding box (AABB) in light space for each cascade. The min and max values of the bounding volume are then used to generate an orthographic projection matrix for the light:

```
light.projection = glm::ortho(boundSplit.pMin.x,  boundSplit.pMax.x,
                              boundSplit.pMin.y,  boundSplit.pMax.y,
                              -boundSplit.pMax.z, -boundSplit.pMin.z);
```

The min and max z values of the bounding volume act as the near and far values of the orthographic projection, culling vertices not within the cascade. The actual shadow map for each cascade is done by setting OpenGL's viewport to match the dimensions of the shadow map (a depth frame buffer object).

These are the results of this technique:



Screen Space Directional Occlusion

Screen space directional occlusion is a technique that approximates global illumination in screen space which makes it easy to insert into a deferred pipeline. The difference between SSDO and the more common SSAO technique is that SSDO takes into account information regarding the direction of incoming radiance whereas SSAO ignores this and simply darkens a cavity if samples are being occluded within a hemisphere. SSDO also can be extended to include a single bounce of indirect light (the implementation used in the project uses the diffuse component of the G-Buffer as a source of direct lighting). SSAO was also implemented, but the results produced by SSDO seemed far better than that of SSAO, so it was removed from the project.

Screen Space Directional Occlusion is described in the paper "Approximating Dynamic Global Illumination in Image Space" by Ritschel, Grosh & Seidel. [3] According to the paper, direct lighting from N sampling directions ω_i uniformly distributed over the hemisphere is computed using this formula:

$$L_{dir}(P) = \sum_{i=1}^{N} \frac{\rho}{\pi} L_{in}(\omega_i) V(\omega_i) \cos\theta_i \Delta\omega$$

Where L_{dir} is the direct radiance, **P** is a 3D position, L_{in} is the incoming radiance, V is the visibility function (is it occluded or visible), and $\frac{\rho}{\pi}$ is the diffuse BRDF (scatter in all directions). Random sampling points are generated within a hemisphere and projected back into screen space. With deferred shading, the world position at a pixel can be retrieved from the position component in the G-Buffer. Samples that decrease in distance from the camera's eye due to this projection with respect to retrieved position are determined to be occluders since the sampling point goes beyond behind the surface (the position stored in the G-Buffer). Conversely, when the distance increases due the projection, then the sample point is determined to be visible as it is above the surface.

For samples that are determined to be occluders, indirect lighting can be implemented by summing the radiance sent from those samples towards the surface. An indirect bounce of illumination is approximated by this formula:

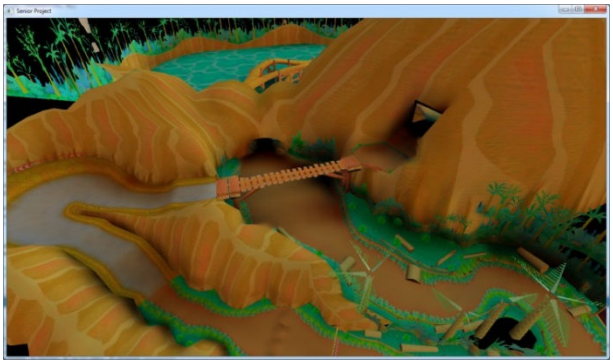
$$L_{ind}(P) = \sum_{i=1}^{N} \frac{\rho}{\pi} L_{pixel}(1 - V(\omega_i)) \frac{A_s \cos\theta_{s_i} \cos\theta_{r_i}}{d_i^2}$$

L_{pixel} is the pixel color for the occluder corresponding to the color extracted from the diffuse component of the G-Buffer when projected into screen space, d_i is the distance between position **P** and the occluder i (with d_i clamped to 1), θ_{s_i} and θ_{r_i} being the angles between the sender/receiver normal and the transmittance direction. A_S is the area for the sender patch. Assuming that the sum of the patch areas are a flat surface inside the hemisphere, we can

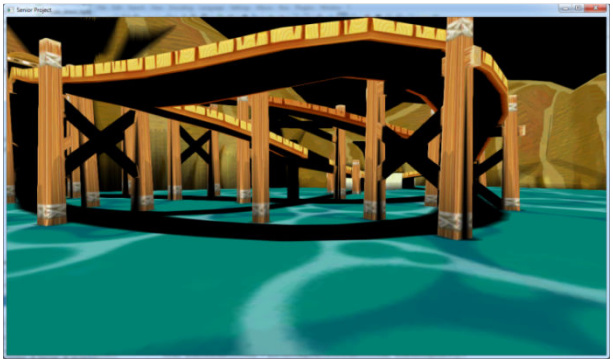
calculate the area each patch covers as $A_S = \pi r_{max}^2 / N$ where r_{max} is the radius of the hemisphere.

Shown below are images showing the various results of the screen space directional occlusion technique:

Plain Scene

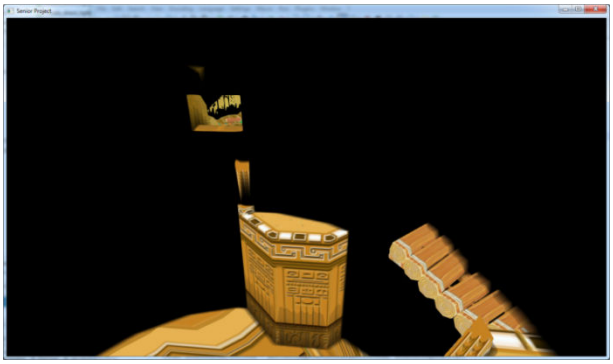


SSDO (No indirect)

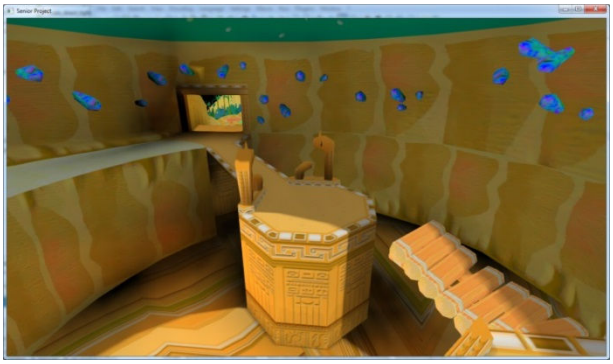


Plain Scene

SSDO + Indirect (Notice blue light reflecting off water)

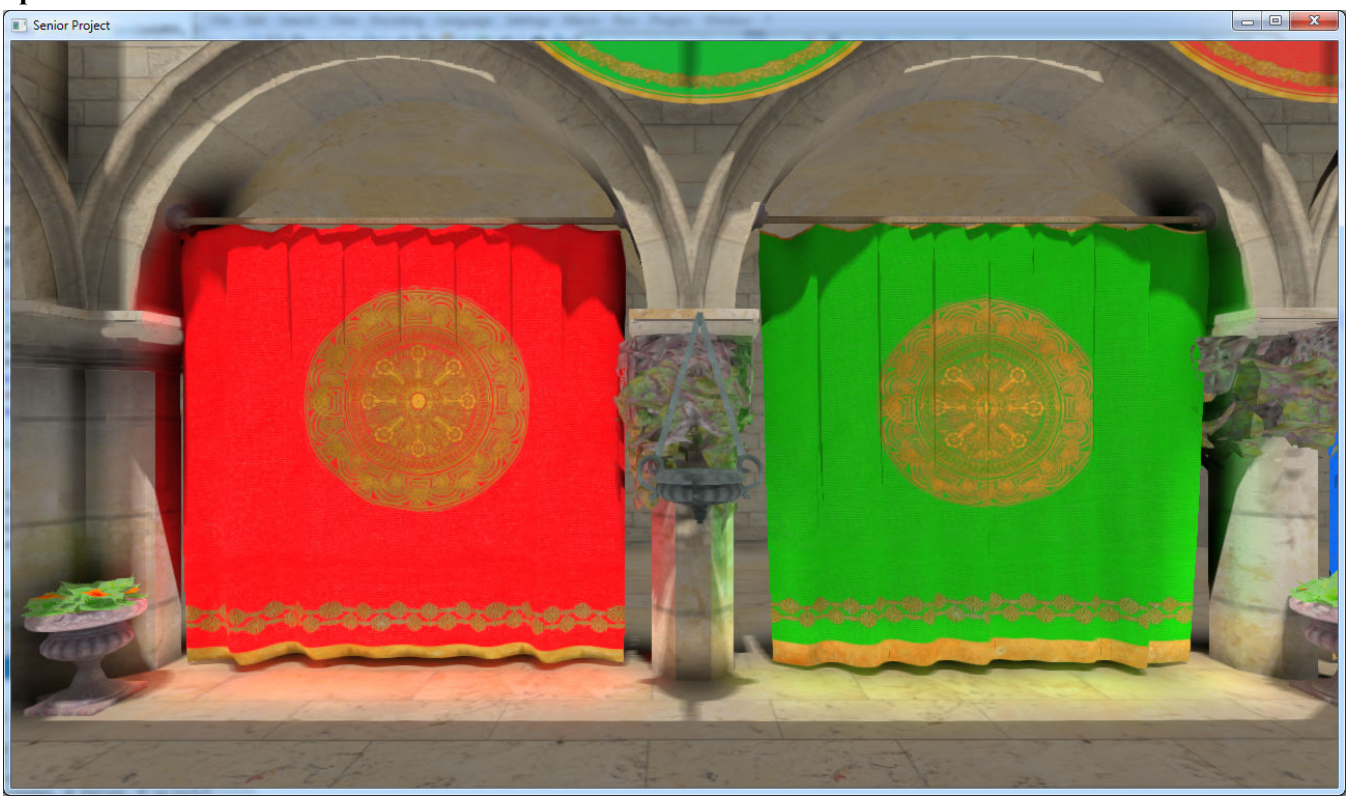


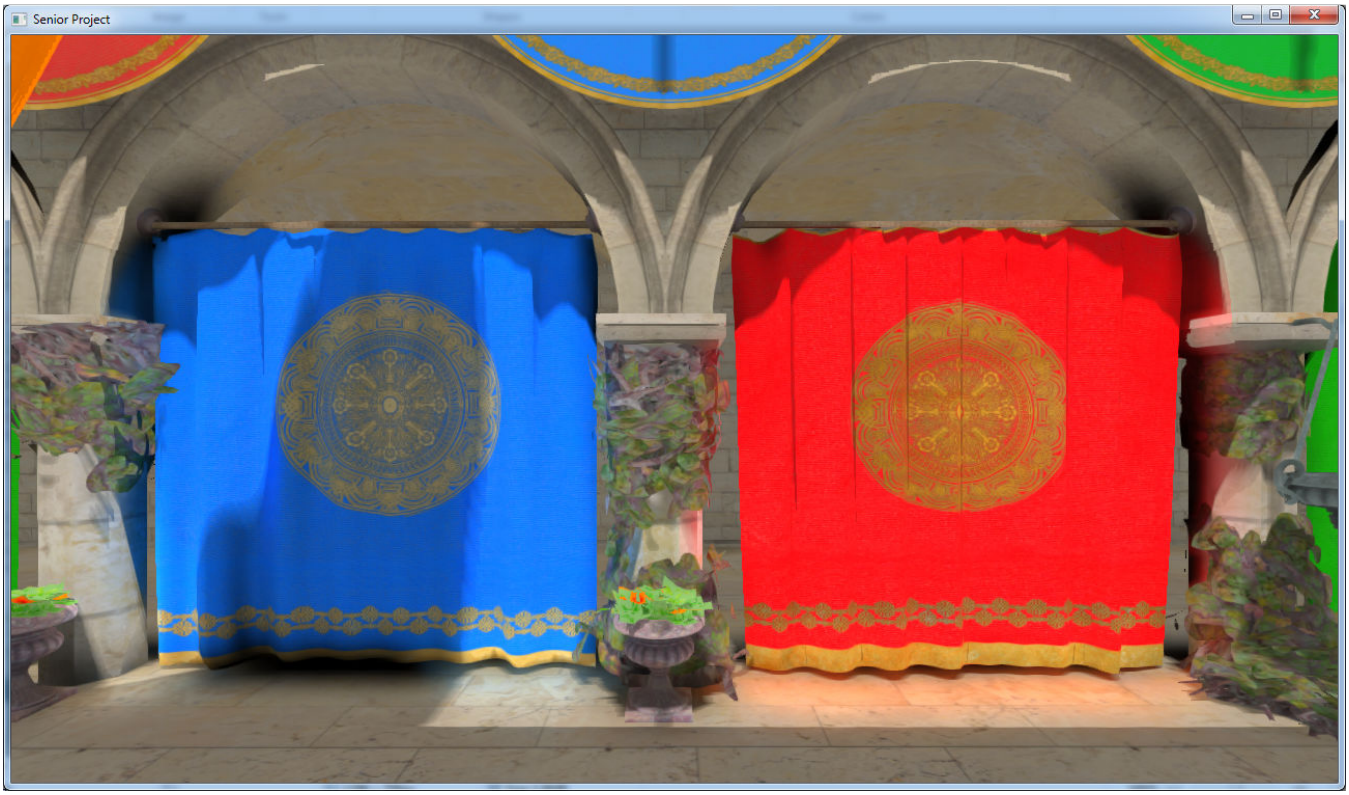
Plain Scene (shadowed)



SSDO + Indirect

Sponza Tests with SSDO + Indirect





Bounding Volume Hierarchy

A bounding volume hierarchy (BVH) has been implemented to help speed up the rendering in conjunction with frustum culling. When meshes are loaded in, an axis aligned bounding box is created for that mesh. Bounding volume hierarchies can also be used to help speed up object collision detection through the use of spatial queries to check against nearby objects instead of checking against every object in the scene.

View Frustum Culling

One optimization technique implemented was frustum culling which works in conjunction with bounding volume hierarchies. For the implementation used in this project, the clip space approach to view frustum culling was used as described in the Lighthouse3D tutorial. [4]

Antialiasing

Due to deferred rendering, hardware antialiasing does not work since the geometry is separate from the lighting. For this project, Fast Approximate Antialiasing (FXAA) was implemented. FXAA was chosen due to its easiness with regards to integrating it into a deferred pipeline and because it is relatively fast compared to other antialiasing techniques such as MSAA. [5]

Conclusion

In this project, a real time rendering engine based on a deferred rendering pipeline was developed. A simple screen space global illumination solution was implemented and is shown to be working. In retrospect, while screen space directional occlusion provides better results than the usual screen space ambient occlusion method, it is very slow and not suitable for older rendering hardware (for example, anything before the AMD Radeon HD 5000 series). For example, the sponza scene was measured running at 22FPS on an AMD Radeon HD 7970 at a resolution of 1024x768 which is not too impressive. Also, as a screen space technique, it is limited to geometry visible and stored in the G-Buffer. Faster and more complete real time global illuminations exist and provide more detailed results. Regarding shadowing, cascaded shadows mapping seems to be a very good technique for getting around aliasing and perspective problems that plague other shadow mapping techniques. Cascaded shadow mapping also seemed to have very good performance. Bounding volume hierarchies and view frustum culling provided a little boost in performance. Finally FXAA, the antialiasing method used in this project, provided a very fast method for antialiasing. However, as mentioned in many sources regarding FXAA, objects and geometry that have detailed textures for shading are blurred/smoothed out resulting in a loss of detail.

Future Work

Global Illumination and Lighting

While Screen Space Directional Occlusion provides an easy way to implement global illumination in a deferred rendering pipeline, it has limitations due to it be being based in screen space. An interesting global illumination algorithm used in CryEngine is a method called Light Propagation Volumes. Another algorithm that has gained recent interest are sparse voxel octrees with cone tracing.

Material Shading

Cook-Torrance has been implemented in the project to provide physically based specular highlights for materials. However additional BRDF models exists that can express additional materials. For example, the Oren-Nayar diffuse model is physically more accurate the Lambert diffuse model used in rendering engine.

Post Processing

More shaders can be implemented to perform additional post processing. With the deferred rendering pipeline, techniques that work in screen space should be easy to integrate into the engine. Examples of possible shaders:

- Lens flare
- Bloom
- Chromatic aberration
- SMAA (Subpixel Morphological Antialiasing)

References

[1] "CryEngine - Visuals." Crytek. <http://mycryengine.com/?conid=8>.

[2] Dimitrov, Rouslan. "Cascaded Shadow Maps." NVidia. <http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf>.

[3] Ritschel, Tobias, Thorsten Grosch, and Hans-Peter Seidel. "Approximating Dynamic Global Illumination in Image Space." MPI Informatik. <http://www.mpi-inf.mpg.de/~ritschel/Papers/SSDO.pdf>.

[4] "View Frustum Culling." *Lighthouse3D*. <http://www.lighthouse3d.com/tutorials/view-frustum-culling/>.

[5] Lottes, Timothy. "FXAA." NVidia. <http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf>.