An OpenGl Desert Scene

Sam Serrels
40082367@napier.ac.uk
Edinburgh Napier University
Computer Graphics (SET08116)

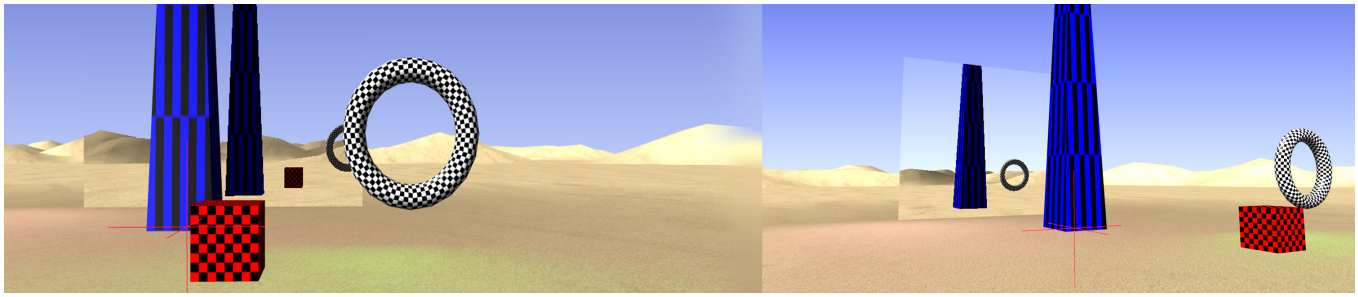

Figure 1: *Project screenshot*

Abstract

This project aims to develop a Real-time 3D graphics scene, with an emphasis on high aesthetic quality. The scene will use a pre-written OpenGL render framework to save development time for producing graphical effects. This project aims to also look into the cutting edge features available in the newest OpenGL standards and investigate the performance benefits of such features.

Keywords: OpenGL, GLSL, Graphics, SSBO, Reflections

1 Introduction

Figure 2: ***Project Inspiration - The Witness*** *- [Thekla, Inc. 2015]*

Project Aims The setting for the project scene is a Desert. Although not a visually busy scene, a desert provides vibrant and harsh lighting, interesting landscape and plenty of opportunity to squeeze more visual fidelity out of a small amount of scene elements. Using a multitude of texture effects, such as bump and parallax mapping, blend maps and level-of-detail, this project plans to bring life to even the most basic of objects, like sand on the ground.

Sky With the wide open vita of a desert plain, the sky takes center stage and is the most important visual cue to selling the realism of the scene. This project aims to have a fully dynamic and procedural sky system, without relying on static textures. This will allow for a time-of day system and full control of elements such as clouds and sun position.

Water The centrepiece of the scene will be a water feature of some kind, providing a contrast against the dry desert and visually interesting element in it's own right. Reflections, waves using distortion maps, refraction, and particle effects will be used to provide realistic looking water.

2 Background / Related Work

Figure 3: ***Spec Ops: The Line*** *- [Yager Development 2012]*

Desert scenes are not uncommon to video games, the simplicity of the landscape was a large bonus to performance limited software. Older games (pre DirectX 9), could get away with a simple ground mesh, a single sand texture and an interesting skybox, and this would be considered a sufficiently detailed scene for the time. See Figure: 4
As the graphical power of computer increased, the challenges to make a realistic desert increased dramatically, the standard of games required more than just a barren wasteland. Buildings, foliage, human characters and interesting landscape were needed and now the scene is just as complex as any other environment. Some modern games have taken up the challenge to model sand behaviour, as a very simple fluid dynamics system to add life to a scene. Sandstorms are a popular feature as they offer the valuable benefit of obscuring level

elements, these elements now do not have to be rendered, therefore increasing performance.

Figure 4: ***Project Inspiration - Guild Wars*** *- [ArenaNet 2005]*

Sky Procedural skys have been used in games for many years, the choice to use them depends on the needs of the game. Games that have aim to have a more living environment are the primary uses for procedural skies, in other games if a static image is sufficient, then a simple texture is used.

Figure 5: ***Procedural Sky - Fallout 3*** *- [Bethesda Game Studios 2008]*

3 Implementation

3.1 Sky

Rendering the sky was achieved by rendering a single screen sized quad to the screen, set at a maximum depth to place it behind all the level elements. Calculations based on the players view matrix and the projection matrix, result in two values representing the top bop and bottom boundary of the players view. These values are a percentage between the horizontal horizon and vertical upwards view, this is used to calculate the colour of the sky at the bottom of the screen and the top. The colours are passed to the shader which interpolates between them, completing the illusion that the player is in a skydome.

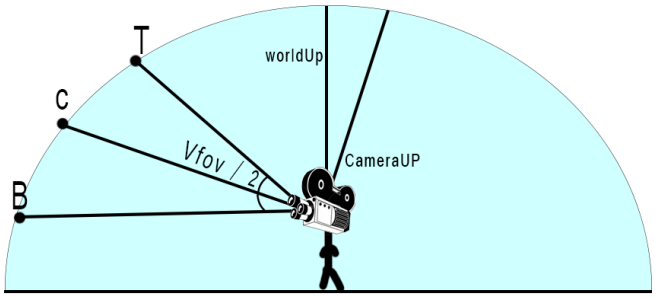

Figure 6: ***Sky Camera Calculations*** *- Distance of sky is infinity*

Listing 1: *Sky calculations*

```
1  // verticle fov = 25.3125deg = 0.441786467 radians
2  float verticleFov = 0.2208932335f; // vfov/2 in radians
3
4  vec3 camview = normalize(cameraTarget − cameraposition);
5  vec3 camUp = normalize(cameraUP);
6
7  float r = atanf(verticleFov);
8
9  vec3 topOfScrnToPlayer = normalize((r ∗ camUp) + camview);
10 vec3 bttmOfScrnToPlayer = normalize((−r ∗ camUp) + camview);
11
12 float topDot = dot( topOfScrnToPlayer, vec3(0, 1.0, 0));
13 float bottomDot = dot( bttmOfScrnToPlayer, vec3(0, 1.0, 0));
```

3.2 Terrain Generation

The distant sand dunes were created by first generating a flat grid of connected polygons, this was looped through and each vertex's y value(height) was modified based on a the output of simplex noise generator. Simplex noise is an optimised version of the Perlin noise generator, both were authored by Ken Perlin, the original Perlin in 1983 and Simplex in 2001.
The generation happens at the start of the program and has many variables to alter the attributes of the generated terrain. As the sand dunes are far away, they are lighted with a simpler, per vertex gouraud model.

3.3 Water

Realistic water is made up of a multitude of components and render passes. At the simplest level, there is the reflected image from the water surface and the image of the area beneath the water surface. Each image will be combined and distorted based on the attributes of the water, the environment, and the position of the player.

Reflection At this stage in the project, only the reflection component of the water renderer has been completed, this results in the mirror effect currently in the scene. Rendering a reflection from a surface requires either using a reflection map for an approximate reflection, or re-rendering the scene from a different point, relative to the players view and position and orientation of the reflected object. For flat surfaces, like water or a mirror, a second render pass is preferred. For complex reflective shapes, reflection maps are almost excursively used.

Rendering Twice The position of the 'Virtual Camera' (Camera B in Figure 7) is calulated by multiplying a reflection matrix by the view matrix of the players camera. The reflection matrix is a matrix that will mirror every position about a plane, the plane being the mirror.

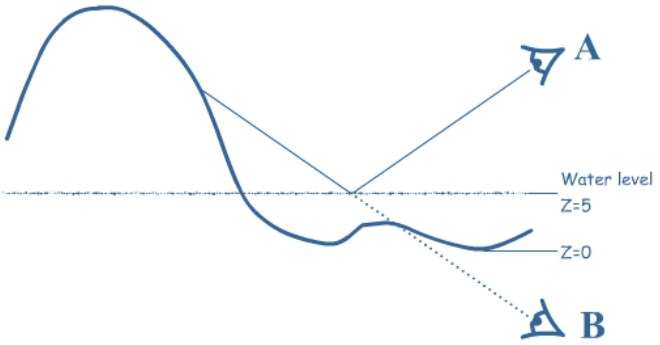

Figure 7: ***Refection Camera position*** *- [Riemer Grootjans 20011]*

Reflected Image Due to the entire coordinate system being flipped, an additional flip matrix needs to be multiplied against the Virtual camera to flip the Y elements of the polygons in the scene are facing the right direction. Alternatively, depending on the placement of the mirror, this can be avoided by simply changing which sides of faces are culled during rendering. More advanced approaches take the form of changing the virtual projection matrix to be an oblique projection with the near plane set to the mirror plane, for easier culling of geometry.

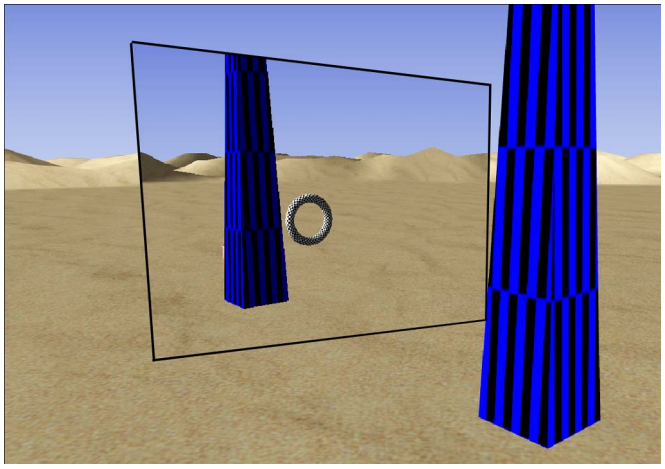

Figure 8: ***Rendered Scene*** *- Mirror Outline overlay for clarity*

Reflected Texture With the reflected image rendered to a texture, this image needs to be applied to the mirror geometry in the main render pass. This process is simple in theory, the section of the image that needs to be used is the same shape as the geometry in the non reflected view. To get this area, the positions of the geometry are transformed by the same matrices used to get the reflected view, and then this data is used as UV coordinates.

Listing 2: *Fragment Shader Reflected UV calculations*

```
1 vec4 reflectedPos = reflected_MVP * vec4(position.xyz, 1.0);
2 vec2 transformedUV;
3 transformedUV.x = reflectedPos.x/reflectedPos.w/2.0f + 0.5f;
4 transformedUV.y = reflectedPos.y/reflectedPos.w/2.0f + 0.5f;
5 vec4 reflectionTextureColor = texture2D (tex, transformedUV);
```

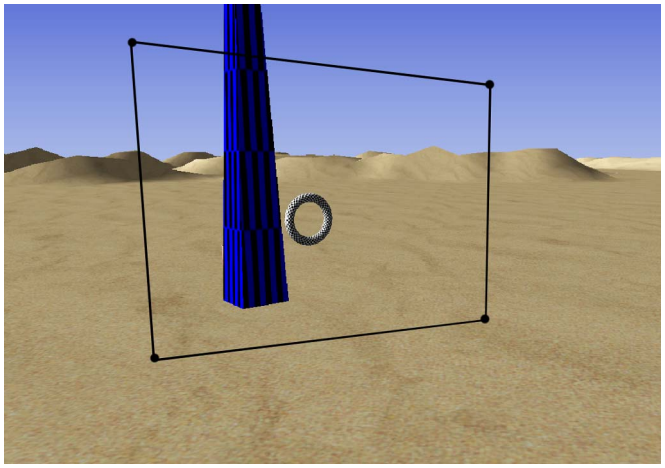

Figure 9: ***Refection Camera View*** *- UV coordinates shown, this is the area that will be rendered on the mirror quad*

3.4 Multiple lighting

A simple approach to multiple lighting in a forward render solution is to send a list of appropriate lights as uniforms to a Shader. This process must be repeated for every light in a scene and for every shader execution. Different methods are available, such as deferred rendering which separates all lighting into a completely separate render pass. The OpenGL standard mandates that all input uniforms must be a predefined size. This means that an upper limit to the amount of lights a shader can accept must be set.

Uniform Buffers A features introduced in OpenGl 3.1 is Uniform buffers. They allow data taht would normally be sent to each shader at render time to be pre-stored in graphics memory. During the render, the program can inform the shader which uniform buffer to load data from. This has the primary benefit of people able to swap between different sets of static uniform data quickly, and that the data only needs to be sent once, and can be read by multiple shaders.

4 Evaluation

Scope The results of the project have been inconclusive in determining the performance benefits of physics optimisation across multiple platforms. The main shortcoming has been the Playstation 3 portion of the project. Due to unforeseen technical problems and the time constraints of the project, the support for the Playstion 3 could not be completed to a level which would result in usable results. Issues with the rendering framework and subtle differences in the maths libraries across the two platforms caused substantially more time spent on fixing code issues than developing features.

Playstation 3 results Unfortunately, optimising the code for the strengths of the Playstation 3, such as subdividing the work to the separate SPU processors was not achieved. Rendering issues caused the final output of the project to be generally unviewable on the Console.

PC results The results of the simulation were more positive on the PC architecture. The simulation runs well and without any major issues, major features detailed in the original design have been omitted due to time constraints. Visually the program is basic, without any graphics embellishments to bring the output closer to the original design vision.

5 Future Work

The primary focus for future work would be to fix and complete the Playstation code to bring it upto par with the PC code, then other features can be worked on. Firstly, dynamic control of simulation variables at runtime, allowing the user to interact with the simulation while it running. The User should be able to adjust parameters such as the size of the world, the attributes of the spacial partitioning, and the ability to create and remove balls at will.
More complex collision shapes would be the next area of work, followed by rendering improvements.

References

ARENANET. 2005. Guild wars.

BETHESDA GAME STUDIOS. 2008. Fallout 3.

RIEMER GROOTJANS. 20011. Reflections. *Accessed: Feb 2015*. www.riemers.net.

THEKLA, INC. 2015. The witness.

YAGER DEVELOPMENT. 2012. Spec ops: The line.