University of Queensland

COSC3000 Semester 1 2015

Model UQ
Computer Graphics Report

Thomas Cranny

42920368

Abstract

TODO

June 5, 2015

Contents

1 Introduction **2**

2 Methods **4**
2.1 Technical Overview 4
2.2 Graphics Environment: WebGL and Three.js 4
2.3 Features 4
2.3.1 Mouse interactivity 4
2.3.2 Texture and bump mapping 5
2.3.3 Dynamic model creation 5
2.3.4 Lighting and shadow casting 6
2.3.5 Animation 6
2.3.6 Constructive geometry 7
2.3.7 GLSL shaders 7

3 Results **10**
3.1 Environment 10
3.2 Texturing 10
3.3 Lighting 11

4 Discussion **13**

5 References **14**
5.1 Technical Resources 14
5.2 External Libraries 15
5.3 Art Assets 15

1 Introduction

The aim of the project was to construct a viable 3D visual representation of the University of Queensland's St Lucia campus. The rationale for undertaking this project was multifaceted; it is a thematic continuation of the previous data visualisation project, and it serves as a useful and relevant application of computer graphics techniques.

Following the initial proposal, the end goal of the project was to create a visually recognisable recreation of the UQ St Lucia campus. In order to accomplish this, multiple computer graphics techniques were utilised to produce the end result.

The previous data visualisation project aimed to analyse the relationships between the faculties, courses, majors, and programs offered by UQ. This project is a continuation of the UQ centric theme, however not much else is continued through into this project.

The rationale of visualising the St Lucia campus in a three dimensional model is to provide a better method of gaining a spatial awareness of the campus' layout and structures. This stems from recalling the unhelpfulness of most UQ maps when trying to navigate the campus for the first time. Due to the odd shapes of many buildings, the presence of many annexes and similarly labelled and named buildings, the ambiguity between roads, walkways, and viable pathways, and the dense arrangement of buildings, sufficiently learning the layout of the campus may even take weeks.

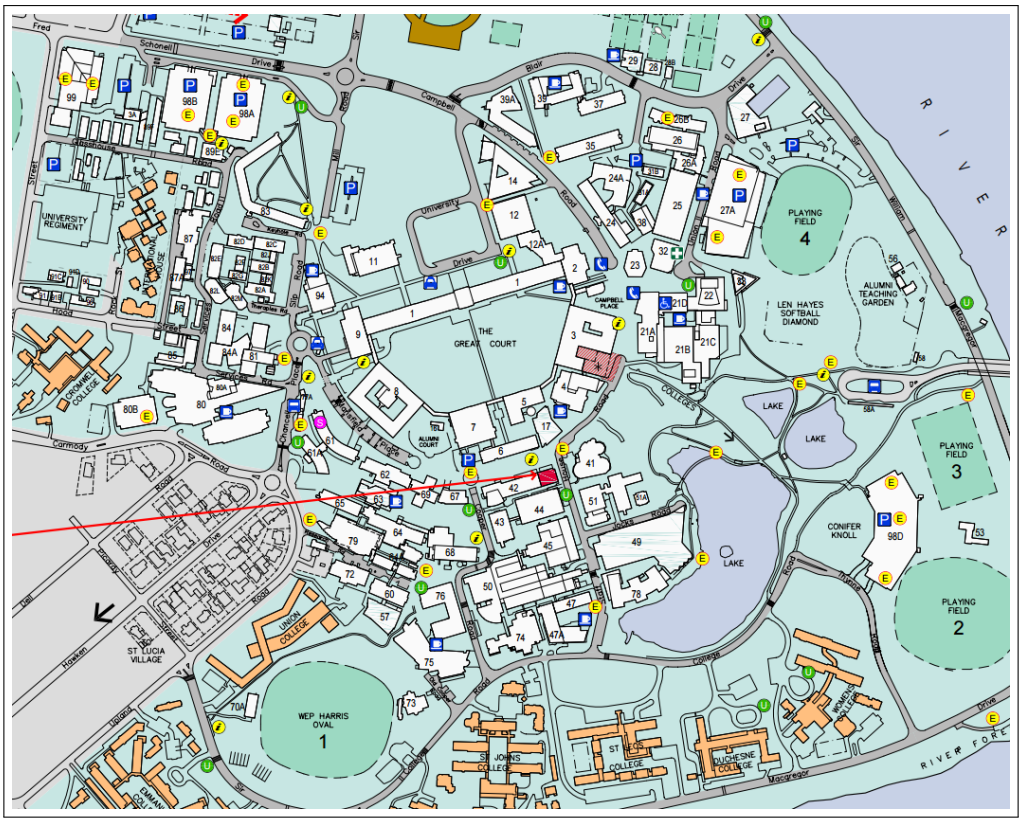


Figure 1: Official University of Queensland St Lucia map [2]. Note the densely packed, oddly shaped and often conjoined layout of the buildings.

Static maps, such as in Figure 1, can have high technical accuracy, but are often convoluted and typically don't convey a good physical model of an environment to a reader. Abstract two dimensional maps typically require a lot of cross checking between the map and the physical surroundings, in order to correlate real landmarks with what the map displays. If structures and landmarks could be better represented to correlate with their real life counterparts more realistically, users will be able to visualise and integrate a locale much more quickly and effectively than attempting to correlate an abstract diagram to their surroundings.

2 Methods

2.1 Technical Overview

The project was implemented using the graphics capabilities of modern browsers, primarily through the three.js JavaScript library, and GLSL shaders. The main features incorporated include:

- Mouse interactivity.
- Texture and bump mapping.
- Dynamic model creation.
- Lighting and shadow casting.
- Animation.
- Constructive geometry.
- GLSL shaders.

2.2 Graphics Environment: WebGL and Three.js

The project was done using WebGL, an implementation of the OpenGL specification used with in modern web browsers. WebGL is designed as a rendering context for the HTML Canvas element [5]. The rationale for using WebGL was based around the portability of web pages, and the ubiquity of web browsers. Equivalent OpenGL projects written in C have more complex and fragile requirements, such as a compilers, library headers, and platform specific binaries; and Python requires similar dependencies and suffers reduced performance in addition.

In order to access the browsers rendering capabilities, the *three.js* JavaScript library was used, as it is well documented, and has an extensive corpus of examples online. It provides a programming environment well suited for graphics programming in JavaScript. While JavaScript is more object orientated than the venerable C bindings for OpenGL, it maintains a close equivalence to native GL instructions, with a cleaned up API and astute use of flexible JavaScript objects[1].

2.3 Features

2.3.1 Mouse interactivity

A major usability feature of the application is the implementation of 'Orbit' controls, which allow users to pan, zoom, and rotate the camera around the scene naturally using a mouse. Mouse interaction is facilitated by registering event callbacks for the main canvas DOM[2] element within JavaScript to fire for different input methods. For example, a function can be registered for a mouse-drag event; should a mouse drag event occur, that function can determine how far the mouse's position has changed with respect to the x and y dimensions, and can manipulate the

[1] '*JSON*' is also often used, it stands for JavaScript Object Notation, and is a useful data serialisation format
[2] Document Object Model, the tree of HTML elements that comprise a webpage.

cameras position to simulate a panning motion.

Earlier versions of the project had a this implemented simply, however later mouse controls were delegated to the better performing and more natural 'Orbit Controls' library [12], which provides simple configuration for $y+$ (always vertical) camera controls, including panning, zooming, and rotating (Added in commit `37bc245`, see reference [1]).

2.3.2 Texture and bump mapping

Textures were used for most of the geometry for the scene in different ways. This includes the buildings, ground, water, and the skybox.

The Ground has the simplest texturing in the scene. It has a grass image applied to it, using repeat S and T wrapping to create a continuous "grassy" surface. A fairly low specular value was used, due to it's lack of reflectivity in real life.

The Buildings in the scene have both a texture and a bump map applied. The bump map is of a simple tessellating brick pattern, and the texture is an image of varying sandstone patterns. The sandstone textures were created by assembling sections of photographs taken on campus specifically for this project. The combination of sandstone and a brick bump map is obviously utilised to simulate the actual sandstone bricks that buildings at UQ are famously constructed from, and produces a believable three-dimensional brick effect for surface of the buildings when inspected.

The Skybox utilises six textures of a cloudy sky, projected onto the interior of a large cube surrounding the scene in order to produce a realistic sky effect. A special purpose built-in shader is used to handle flipping the normals of the skybox cubes face, so the inside surface of the cube is visible instead of the outside. The shader also serves to illuminate the sky textures so they are still visible at great distances, and map the images to the correct location and rotation so that it appears seamless.

The Water of the UQ lakes and Brisbane River utilise a texture of caustic water map. The water however is a special case, as the texture is processed and manipulated via a GLSL shader, which provides a subtle rippling or wavy effect on the water surface. See 2.3.7 below for more information on the shaders behaviour.

2.3.3 Dynamic model creation

In order to specify and generate the buildings and water bodies of the model efficiently, a dynamic approach to model initialisation was taken. The process involves defining an arbitrary set of anticlockwise points for the "footprint" of each building and lake. This information is stored in the `data.js` JavaScript file, and also defines for each building the position, rotation, name, building number, and height.

In the case of buildings, each in the dataset is iterated, and a two-dimensional shape is constructed from the array of footprint points. Then, that polygon is extruded vertically by a factor of that buildings height. This allowed for moderately fast input of the buildings and water bodies

into the model, as only a position, height (for buildings), and small set of points was required to define each feature.

The initialisation of the water bodies is more complicated, as first geometry is subtracted away from the terrain to allow room for a water surface to appear below the level of the terrain, and then a shader is initialised to produce the rippling water effect on the water surface meshes. See 2.3.6 and 2.3.7 for further information on the geometry and shaders respectively.

2.3.4 Lighting and shadow casting

The scene is lit using a variety of techniques. Firstly, there is a low level ambient light used. This provides a small amount of illumination to all objects in the scene. This serves to lessen the harsher shadows of direct lighting, and ensures darker parts of the scene are visible and not completely black.

The scene also has what is called "hemispherical" illumination applied, which is a useful technique for simulating light sources at optical infinity with an underlying plane present. In effect, this means it is well suited to simulating sunlight that also artificially reflects the colour of the ground. This was used to provide a soft yellow sunlight effect from above the scene, and a subtle green diffuse reflection coming from the bottom of the scene, to simulate a diffuse reflection of the grass. This is best observed on the downward facing normals of the bump mapped bricks, which the buildings are textures with (see figure `TODO`).

The final light source of the scene was a directional light, which was emitted from above and off to the side of the model. This light was centred within a small yellow sphere with an emissive yellow colouring to simulate a sun. The light and sphere also rotated around the scene, and moved up and down as well. When running, buildings would therefore constantly cast a moving shadow that also changed length with the rotation and angle of the "sun" object.

2.3.5 Animation

The project incorporated a few minor animation techniques, primarily the movement of the sun and related directional light, and the "growing" animation of the buildings when the model is first loaded.

When the web page containing the model program loads, all of the buildings appear to "grow" out of the ground over the course of the first five seconds to their full height, regardless of how many levels the structure contains. This effect is performed by scaling the y dimension of each building by a factor of the time the animation has been running divided by 5 seconds. This is obviously stopped after 5 seconds have passed, so the buildings do not continue to grow past their intended heights.

The other form of animation present is the rotation and bobbing of the sun mesh and light source. The small sphere representing the sun, continually rotates around the model, and changes its height according to a periodic function. This animation exists to convey the dynamic lighting and shadowing effects utilised to the user.

2.3.6 Constructive geometry

In order to simulate the appearance of bodies of water more accurately, a volume below the area of each lake[3] covered was removed using the subtraction method of constructive solid geometry. In order to accomplish this, a polygon is created within the footprint of the lake. Similarly to the buildings, the base polygon is extruded in the y dimension to create a three-dimensional volume mesh, however in this case it is by a small amount downwards. Using a CSG library, that mesh is then subtracted from the ground mesh it overlaps with, providing a cut out depression in the shape of the lake. Then a two-dimensional clone of the initial polygon mesh is put in place, slightly lowered and textured to act as a water surface.

2.3.7 GLSL shaders

Shaders are independent programs that can be compiled and run when a graphics program is initialised. The two major types are vertex and fragment shaders, which respectively are able to modify vertices and pixel colouring data of their input at a very late stage of the rendering process. Shaders are therefore a component in the utilisation of a programmable pipeline, and allow for very powerful effects to be made relatively simply.

In this project, a vertex and a fragment shader were written in GLSL, the OpenGL Shader Language, and used to manipulate the surfaces of the bodies of water. The main effect achieved was to provide a ripple effect on the surface of the water, and move and manipulate a texture to simulate the appearance of water in a better fashion.

The two shaders used are quite small, particularly the vertex shader, as the vertices of the perimeter of the lakes are not manipulated. The fragment shader however manipulates a texture of water, by using an image of Perlin noise, and a offsets it by a function of time and the red and blue colour channels of the noise map. The image textures and the time value are given to the vertex shader as `uniforms`, which are initialised by the parent program for each render pass. See Figure 2 for a screenshot of the end result, and Listing 1 to view the GLSL code used to acheive the effect.

[3]The Brisbane river is also included as a body of water, the term "lake" is just used for simplicity.

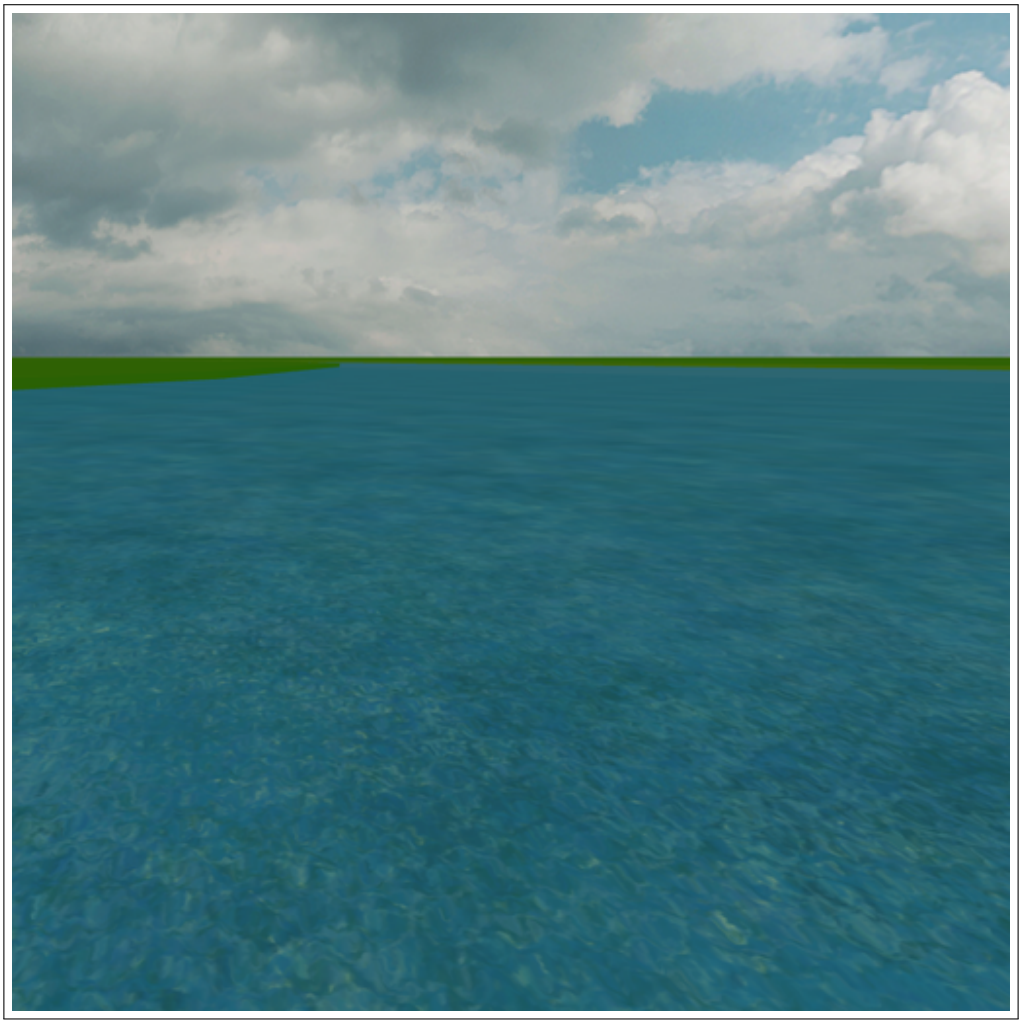


Figure 2: Screenshot of water in the "Brisbane River". When running, the water surface appears to ripple and shimmer semi-realistically, because of a fragment shader.

```
// Uniform variables, provided by parent program.
uniform sampler2D waterTexture;
uniform float waterSpeed;
uniform sampler2D noiseTexture;
uniform float noiseScale;
uniform float alpha;
uniform float time;

varying vec2 vUv; // UV coordinates, shared by prior initialised vertex shader.

void main() {
    // Scale S and T coordinates for better wrapping.
    vec2 repeats = vec2(0.05, 0.05);
    // Create a vector to offset the texture by based on time.
    vec2 uvTimeShift = vUv + vec2(-0.5, 1.5) * time * waterSpeed;
    // Sample the noise map, by time manipulated coordinate.
    vec4 noiseGeneratorTimeShift = texture2D(noiseTexture, uvTimeShift);
    // Determine shift by red and blue channels of noise sample.
    vec2 uvNoiseTimeShift = vUv + noiseScale *
        vec2(noiseGeneratorTimeShift.r, noiseGeneratorTimeShift.b);
    // Calculate the final colour for the pixel, by sampling from the original.
    vec4 baseColor = texture2D(waterTexture, uvNoiseTimeShift * repeats);

    baseColor.a = alpha; // Set the pixel to the specified transparency.
    gl_FragColor = baseColor; // Update the output pixels colour.
}
```

Listing 1: The GLSL source of the water fragment shader. Lines 22-42, `index.html`

3 Results

3.1 Environment

When the web page containing the model loads, a recognisable representation of the University of Queensland's is visible.

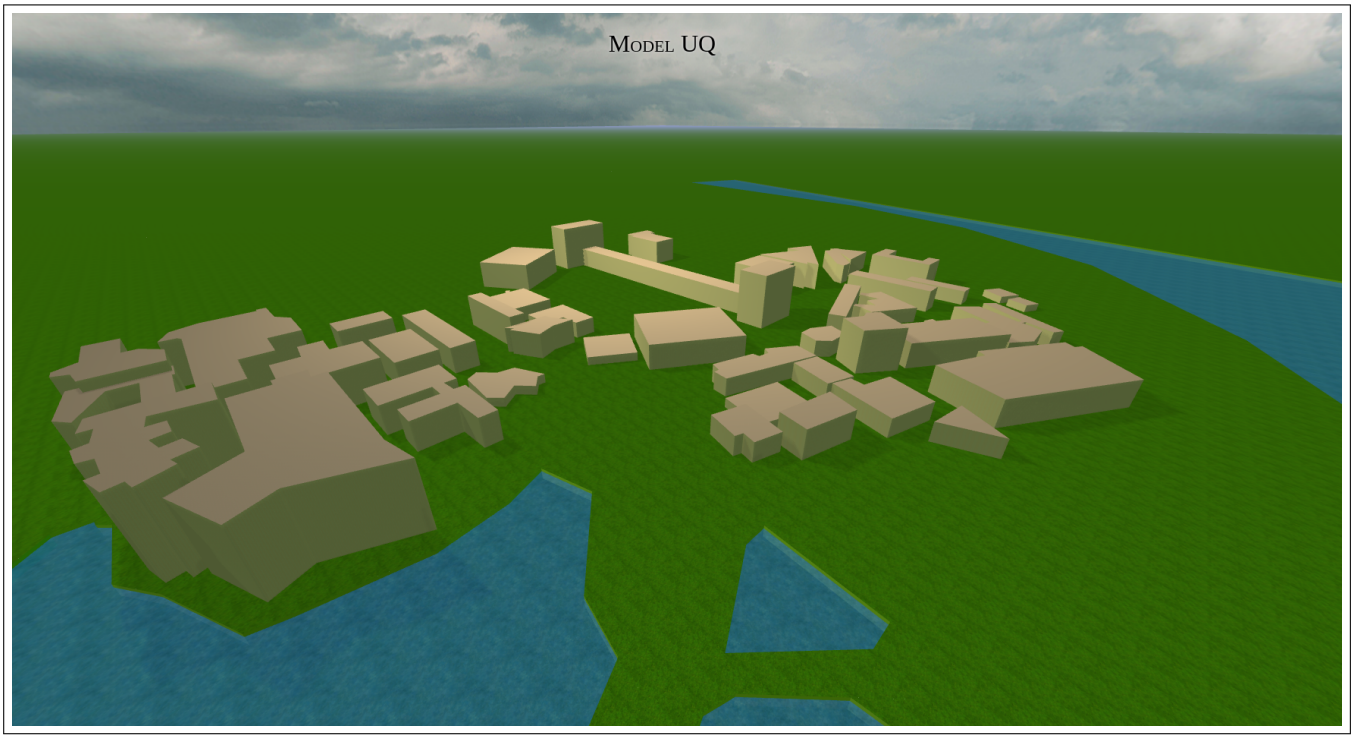


Figure 3: Screenshot after the web page loads.

It is clear that there are many buildings in a location surrounded by a couple of bodies of water, including what appears to be a river wrapping around. Should the user be familiar with the St Lucia Campus, the Forgan Smith building and the Great Court are particularly recognisable as major features identifying the campus. Many of the structures are also immediately recognisable, such as the distinct wedge shape of the Advanced Engineering Building overlooking the lakes, and the Michie and Duhig tower buildings bookending the Forgan Smith.

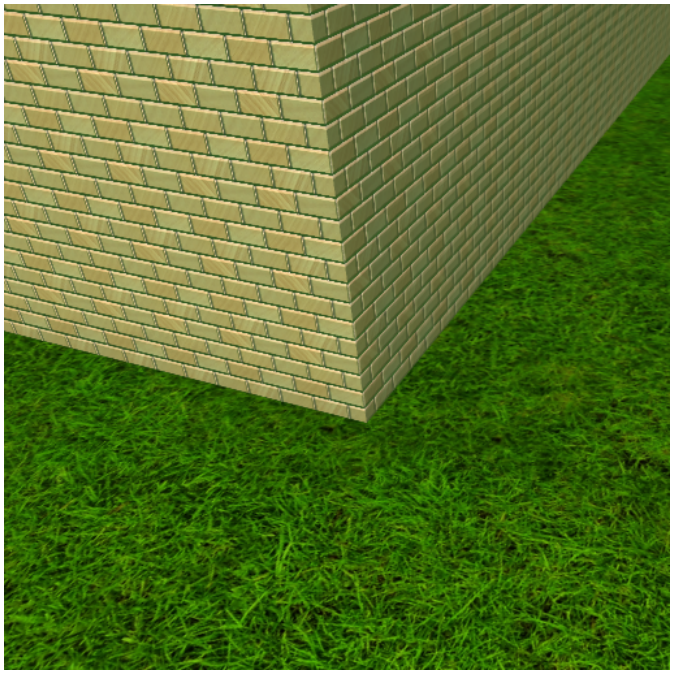
It is also evident the water bodies are recessed into the ground, with a defined depth, below the level of the water. The other major striking feature evident early on is the presence of a realistic sky, normally coloured and with clouds in the distance.

Overall, the scene conveys an understandable representation of a set of buildings, with multiple bodies of water in the vicinity quite successfully. A user observing this model should be able to immediately get a sense of orientation and the positioning and heights of the buildings present.

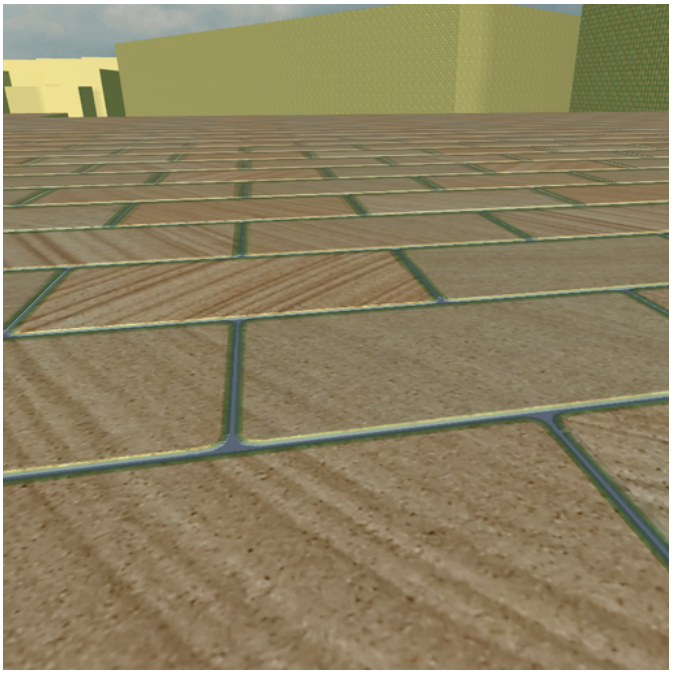
3.2 Texturing

While reasonably subtle from long distances, it becomes much clearer upon inspection that the grass, buildings, and water are all textured (see figure Figure 4a). When zooming up on the surface

of the buildings, the bump mapping used becomes very apparent, and produces a very believable and clear representation of sandstone bricks (see Figure 4b).



(a) Example of the sandstone and grass texturing.



(b) Bump mapping in effect. Note the occluded edges of bricks further back.

Figure 4: Texturing screenshots.

3.3 Lighting

The lighting of the scene greatly improves the quality of the model. Having moving shadows, and correctly lit surfaces is very useful for the users depth perception and the believability of the model.

If the scene is observed for even a small length of time, it becomes apparent that the shadows are moving around the buildings in a circular fashion, and also changing lengths. The shadows cast by the buildings definitely contribute to improving the quality of the scene.

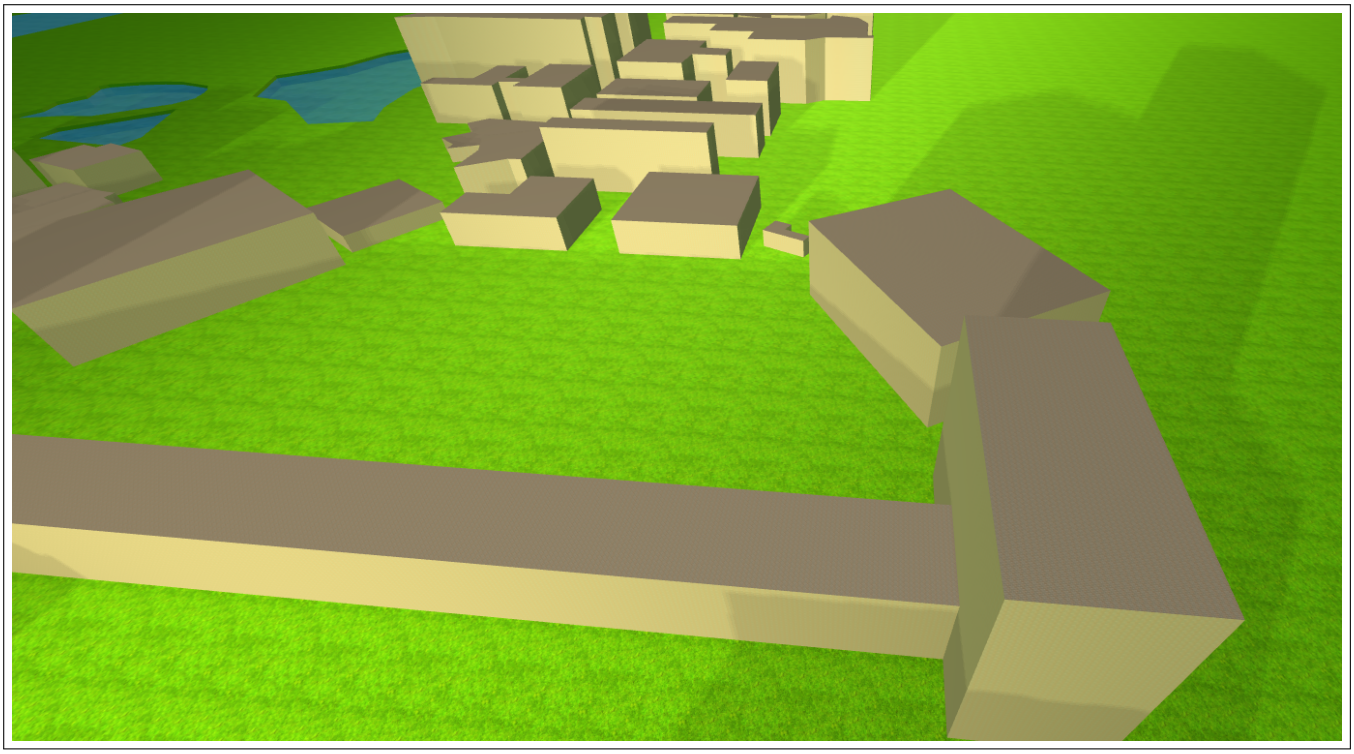


Figure 5: Screenshot of shadows being cast by buildings around the great court, with the sun at a low angle.

4 Discussion

Here be dragons!

5 References

[1] Thomas Cranny *Computer Graphics Project - source code repository*
Accessed online at: https://github.com/kamikai/cosc_project2
On 3/May/2015

[2] The University of Queensland *St Lucia Map*
Accessed online at: https://www.uq.edu.au/maps/pdf/StLuciaMap.pdf
On 1/June/2015

5.1 Technical Resources

[3] Threejs.org *Online Documentation*
Accessed online at: http://threejs.org/docs/
On 1/June/2015

[4] Threejs.org *Online Examples*
Accessed online at: http://threejs.org/examples/
On 1/June/2015

[5] Khronos Group *WebGL Specification - Version 1.0.3*
Accessed online at: https://www.khronos.org/registry/webgl/specs/1.0/

[6] Jerome Etienne *Let's do a skybox* - August 2011
Accessed online at:
http://learningthreejs.com/blog/2011/08/15/lets-do-a-sky/
On 1/June/2015

[7] Roman Liutikov *Skybox and environment map in Three.js* - August 2013
Accessed online at:
http://blog.romanliutikov.com/post/58705840698/skybox-and-environment-map-in-three-js
On 1/June/2015

[8] Jerome Etienne *Casting Shadows* - January 2012
Accessed online at:
http://learningthreejs.com/blog/2012/01/20/casting-shadows/
On 1/June/2015

[9] Jerome Etienne *Constructive Solid Geometry with csg.js* - January 2012
Accessed online at:
http://learningthreejs.com/blog/2011/12/10/constructive-solid-geometry-with-csg-js/
On 1/June/2015

[10] Lee Stemkoski *Animated Shaders in Three.js* - May 2013
Accessed online at:
http://stemkoski.blogspot.com.au/2013/05/animated-shaders-in-threejs-part-2.html

[11] Paul Lewis *An Introduction to Shaders* - April 2012
Accessed online at:
https://aerotwist.com/tutorials/an-introduction-to-shaders-part-1/

5.2 External Libraries

[12] Threejs.org *Orbit Control Example*
Accessed online at:
http://threejs.org/examples/misc_controls_orbit.html
On 1/June/2015

[13] Evan Wallace *Constructive Solid Geometry*
Accessed online at: http://evanw.github.io/csg.js/
On 1/June/2015

[14] Ricardo Cabello *Performance Statistics Monitor*
Accessed online at: https://github.com/mrdoob/stats.js/
On 1/June/2015

5.3 Art Assets

[15] OpenGameArt.org - *Skybox Textures*
Accessed online at: http://opengameart.org
On 1/June/2015

[16] Threejs.org - *Grass Texture*
Accessed online at: http://threejs.org/examples
On 1/June/2015

[17] Thomas Cranny - *Sandstone texture & bump maps*
Source images taken May 2015