OpenGLTest

Shaneel Sharan

Last Updated: August 29, 2014

1 Introduction

OpenGLTest is a C++ program I've been writing to learn more about OpenGL and graphics programming in general. This program uses OpenGL 3.3, so the graphics pipeline is programmable through the use of shaders. Because of this, however, the program will only run on computers with graphics cards that support OpenGL 3.3 or higher. The complete source code for this project is available at https://github.com/sksharan/OpenGLTest.

Currently, this program offers the following features:

- Implementation of the Phong reflection model using point lights
- Object creation via Wavefront OBJ files (files that encode information about 3D models)
- Terrain creation via height maps (greyscale images that encode height information)
- Generation of procedural, multi-textured landscapes using Perlin noise
- Procedurally-generated grass that renders only within a certain distance of the user and only on appropriate surfaces (e.g. grass won't grow on a steep rocky cliff)
- Creation of Axis-Aligned Bounding Boxes (AABB) and Oriented Bounding Boxes (OBB) for all objects
- Users can generate a set of predefined objects through simple key presses
- Two movement modes: users can freely fly around the scene or simulate walking on the landscape itself
- Settings of one object (e.g. visibility, lighting, scale, position) can be modified independently of all other objects in the scene
- Objects can be selected by clicking on them with the mouse cursor (this mouse-picking is done via an algorithm that checks for ray-AABB intersections)

When the program is initialized, the user starts at the origin of the world, with a procedurally-generated landscape below them that is 256 x 256 vertices in dimension. The following are the keyboard controls:

ESC: Closes the window and terminates the program.

W, A, S, D: Move forward, left, backward, and right, respectively.

M: Toggles mouse-look mode on/off. If mouse-look is enabled, then the program takes control of the mouse and forces it to the center of the window, allowing the user to look around. If mouse-look is disabled, then the user has control of the mouse. Regardless of whether or not mouse-look is enabled, users can always click on an object in the scene to select it (AABB's and OBB's cannot be selected directly, however).

Q, E: Select the previous/next object on the list. Whenever an object is created, it is added to the list of objects that can be selected. The name of the newly selected item is visible through the console window.

R: Change the rendering mode of the currently selected item. There are two rendering modes: "textured" and "normal." The former shows the object when it is properly textured. The latter displays the normals of the object, where all normals are in eye/camera space.

H: Toggles the visibility of the currently selected object. The object can still be selected even when it is hidden.

J, K: Toggles the visibility of the currently selected object's AABB or OBB.

L: Toggles whether or not the currently selected object is influenced by lighting calculations. If it is not, then the object is rendered as if there are no lights in the scene.

T: Toggles wireframe rendering on/off. This affects all objects in the scene.

Y: Toggle movement modes. The user can freely fly about the scene or move as though they are walking on the landscape. Because of the method by which the user's height is determined in the latter mode, the user can run off the end of the landscape without crashing the program.

Z: Change to translate mode. The current object can then be translated using the ARROW KEYS, LSHIFT, and SPACEBAR. Translation occurs in terms of world space.

X: Change to scale mode. The current object can then be scaled using the LEFT/RIGHT ARROW KEYS.

C: Change to rotate mode. The current object can then be rotated using the ARROW KEYS. Rotation occurs in terms of world space.

1: Generates a checkerboard-textured square (a `RenderableObject`) at the user's current location.

2: Generates a tower object (an `OBJObject`) at the user's current location.

3: Generates a `HeightmapObject` at the user's current location. A `HeightmapObject` is a grid-based object whose height values at each vertex are determined by a grayscale "height map" image.

4: Generates a `PerlinHeightmapObject` at the user's current location. They are similar to `HeightmapObject`s except that the height values are determined by a Perlin noise equation.

2 Libraries

The following are external libraries used in the development of this program:

Simple DirectMedia Layer (SDL) 2.0
SDL (https://www.libsdl.org/) creates the OpenGL 3.3 "context" and is used to handle keyboard and mouse input from the user.

OpenGL Extension Wrangler Library (GLEW)
GLEW (http://glew.sourceforge.net/) allows access to a wide array of additional OpenGL functions required for shader-based rendering.

OpenGL Mathematics (GLM)
GLM (http://glm.g-truc.net/0.9.5/index.html) is a mathematics library used to simplify vector/matrix calculations. It is closely based on GLSL and provides other useful functions to the OpenGL programmer.

Simple OpenGL Image Loader (SOIL)
SOIL (http://www.lonesock.net/soil.html) provides an easy way to load images for use in OpenGL as textures.

libnoise
libnoise (http://libnoise.sourceforge.net/) is used to generate Perlin noise as well as other types of coherent noise. Perlin noise is used in this program to create mountainous procedurally-generated terrain (see the `PerlinHeightmapObject` class).

3 Main Function

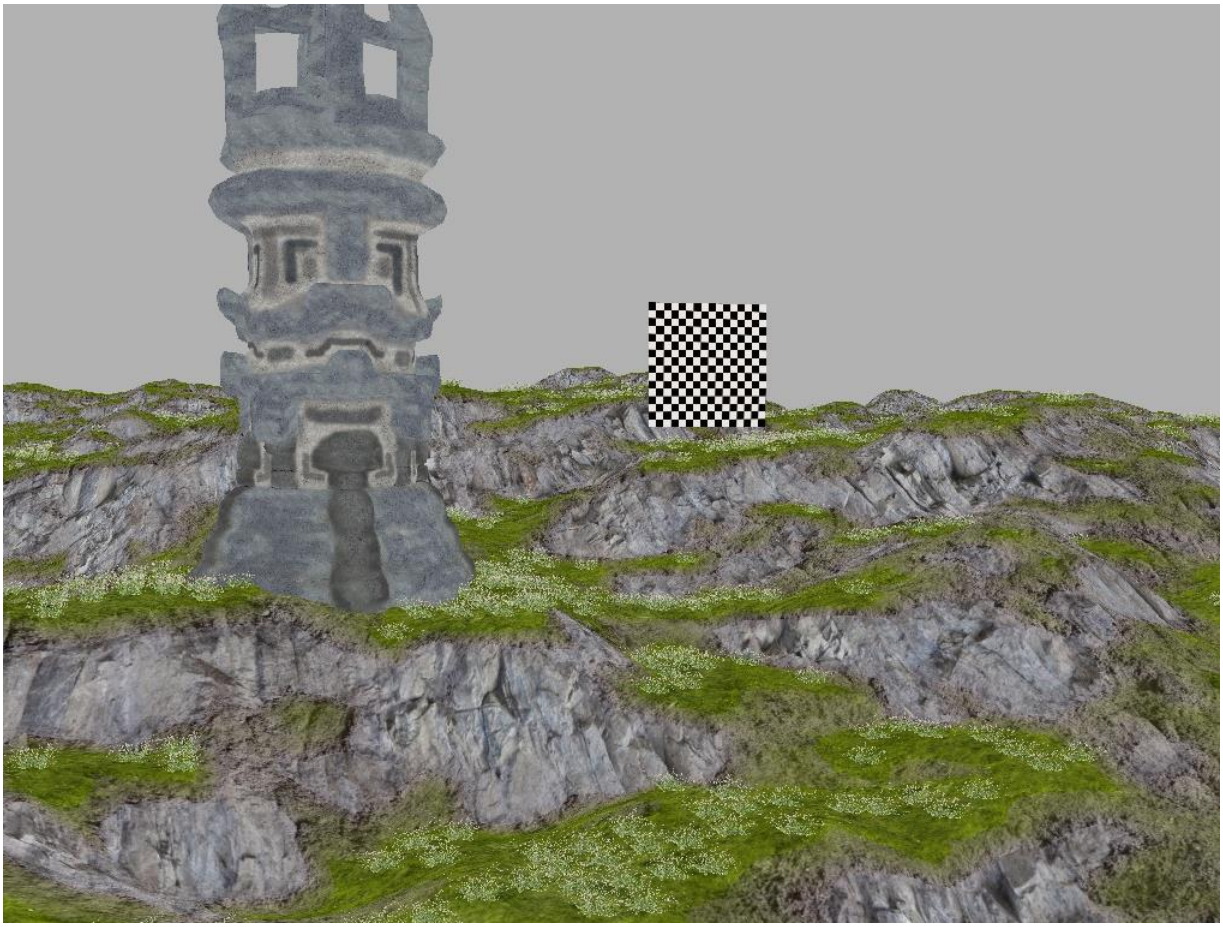
The `main`() function gives a high-level overview of what happens as the program executes. In pseudocode, it looks like the following:

```
int main(int argc, char** argv) {
    initialize SDL
    initialize GLEW

    enable depth buffering, blending, etc.

    create the OpenGL shader and program objects

    set up the sampler variables in GLSL

    set up the perspective, model, view, and normal matrices
    update the corresponding uniform variables in GLSL

    create a Scene object to keep track of all objects to render
    create objects to render and add them to the Scene
    set up the "landscape" object which the user can walk on

    create the point lights

    initialize the mouse cursor

    while (program should still run) {
        handle keyboard input
        handle mouse input
        render the Scene
    }

    delete all objects that were created

    de-initialize SDL

    return 0
}
```

The actual implementation of this function can be found in `Main.cpp`. The first half of the function initializes the dependencies and sets up OpenGL so that rendering will occur properly. A `Scene` object is then created, which simply contains a list of objects to render and a few utility functions for selecting objects in that list. We create objects (in this case, just the landscape that the user starts on) and add them to the `Scene`. We then register the landscape object as an object which the user can walk on (the source code does this by using a class called `LandscapeManager`, which other parts of the program reference if

they need to know about the walkable landscape). We then create as many point lights as needed and set up the mouse position.

The while loop is this program's main loop and currently doesn't account for frame rate. In the main loop, we handle any keyboard and mouse input that the user provides and then call the rendering function. When the rending function is called, we provide it the `Scene`, and the render function renders all the objects in that `Scene`. The advantage to having a `Scene` class is the multiple `Scene`s can be created, each with their own configuration of objects. We can then select which `Scene` to render at any given time.



The tower, checkerboard square, and landscape are all a part of single `Scene` object. By passing this `Scene` object to the rendering function, we get the image above.

After exiting the main loop (which occurs when the user presses the ESC key), the program cleans up and then returns o to indicate success.

4 Rendering Objects

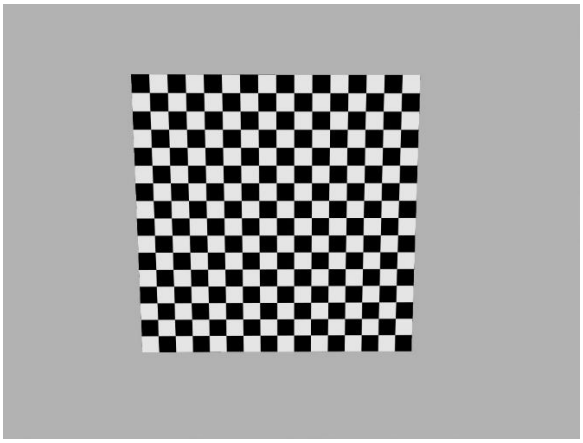
All objects that are rendered to the screen are `RenderableObject`s or children of `RenderableObject.` The class hierarchy for these objects looks as follows:

```
RenderableObject
        OBJObject
        HeightmapObject
                PerlinHeightmapObject
        AABBObject
        OBBObject
```

Whenever one of these types of objects is created, we add it to a `Scene` object, which contains a vector of all the objects to be rendered. Below is a brief overview of each of the different types of `RenderableObject`s.

`class RenderableObject`

The base class to all objects in the program that can be rendered to the screen. When making a basic `RenderableObject`, the user must manually supply the vertices, texture coordinates, normals, and indices of the object.



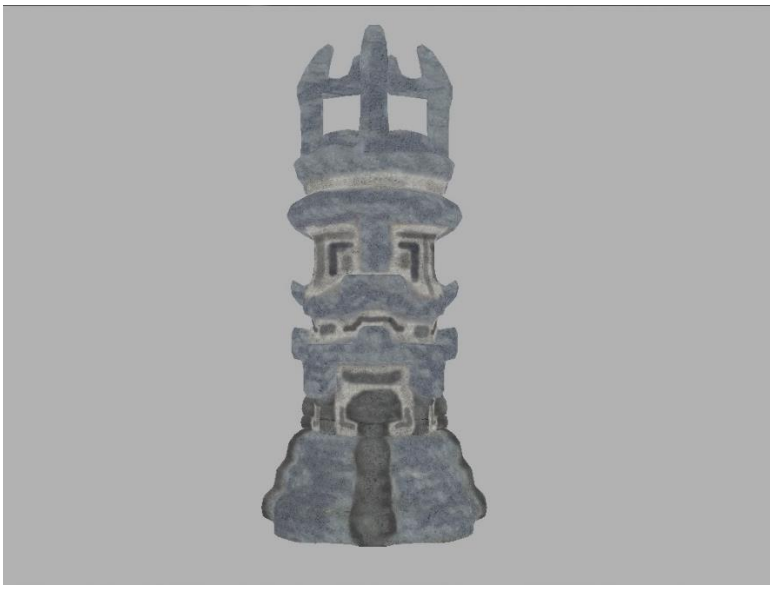
A basic `RenderableObject` made up of 4 vertices. The code to generate this quad can be found in `Objectgenerator.cpp`, under the `genTestSquare()` function.

`class OBJObject :: public RenderableObject`

`OBJObject`s are `RenderableObject`s whose data comes from Wavefront OBJ files. The user supplies the name of the OBJ file, and `OBJObject` parses the data, generating the vertices, texture coordinates, normals, and indices for the object. The program makes some simplifying assumptions about the input OBJ file. For instance, every face must be defined as follows:

```
f vert1/tex1/norm1 vert2/tex2/norm2 vert3/tex3/norm3
```

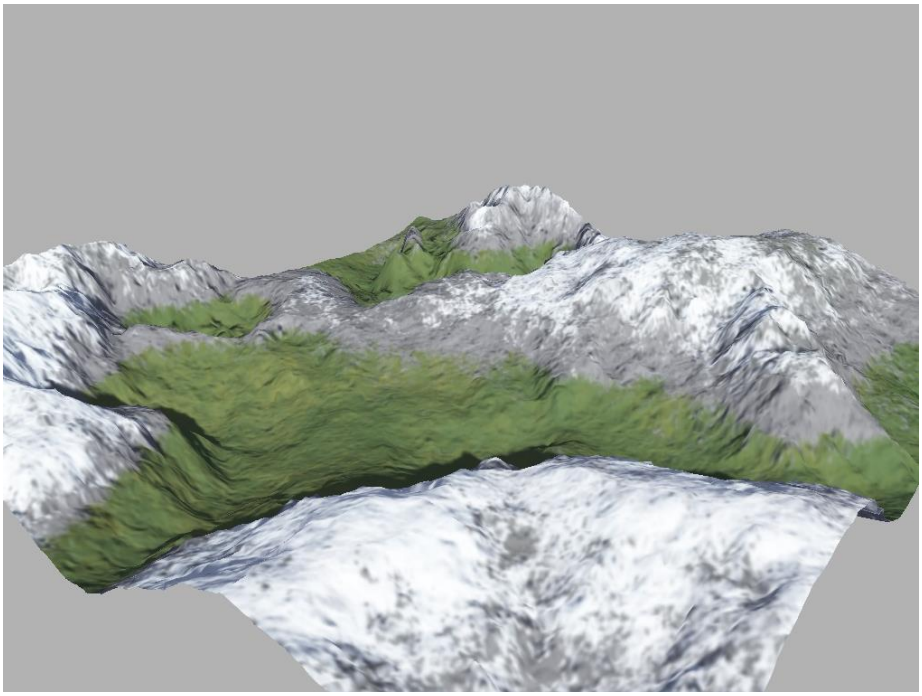
Also, only triangles (not quads) can be defined in the file.



The tower above was created in Blender and then exported as a Wavefront OBJ file. It can found in `obj/tower.obj` *and the associated texture can be found in* `textures/tower.jpg`

`class HeightmapObject :: public RenderableObject`

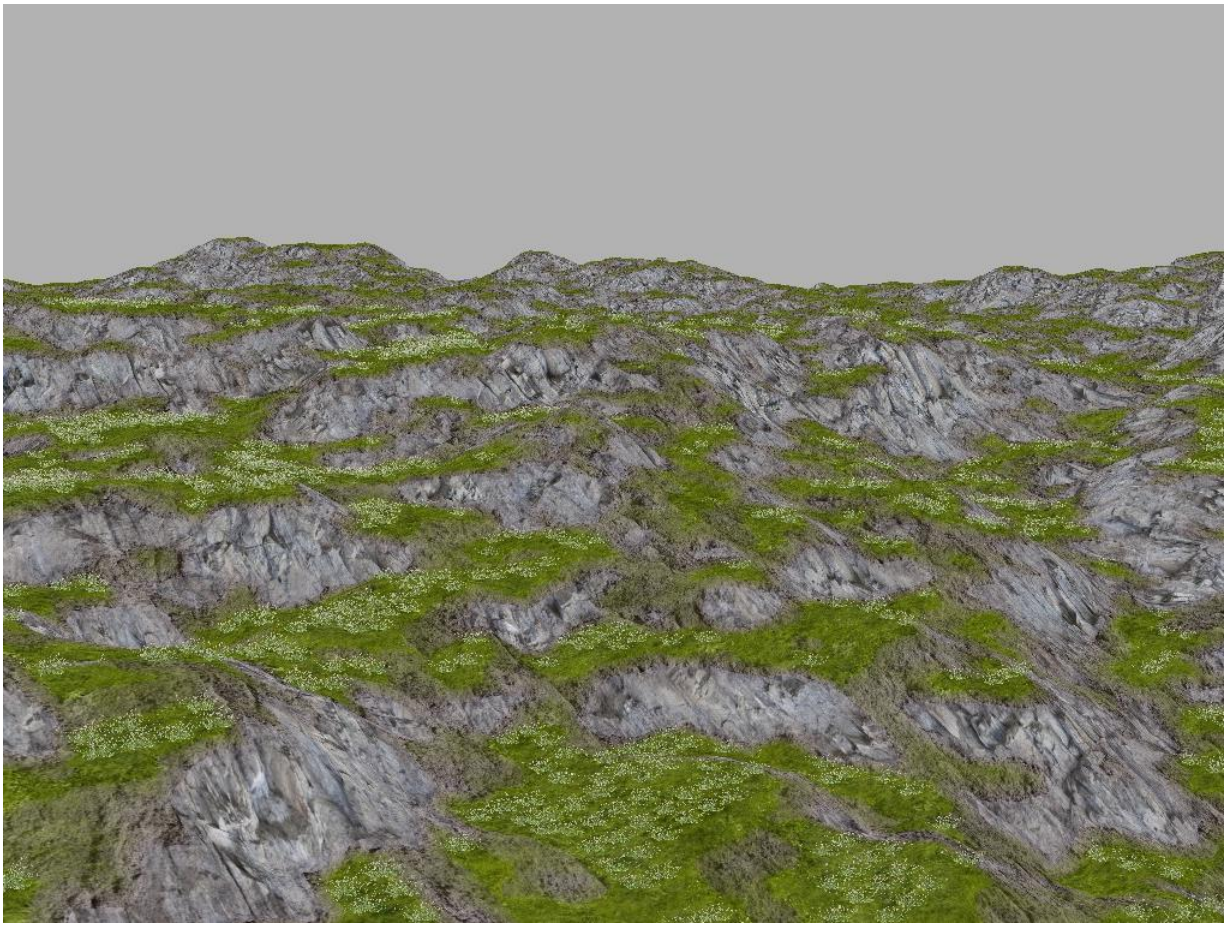
A `HeightmapObject` is a grid-like object that extends in the positve x and negative z directions. The height of each point on the grid is given by a greyscale image called a "height map." The user can specify the position of the `HeightmapObject,` the spacing between vertices on the grid, and the amplitude of the height values (determining how high the mountains will be).



A 256 x 256 `HeightmapObject`*. The heightmap image and the texture used in the image above can be found at* *http://www.chadvernon.com/blog/resources/directx9/terrain-generation-with-a-heightmap/*

`class PerlinHeightmapObject :: public HeightmapObject`

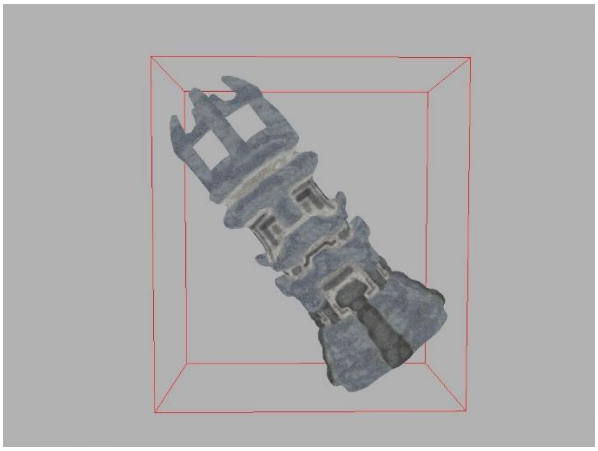
A `PerlinHeightmapObject` is a `HeightmapObject` whose height values are determined from a Perlin noise equation instead of from a greyscale image. `PerlinHeightmapObject`s are used as "landscapes" in the program, so the user can walk on them. Specialized shaders are used for `PerlinHeightmapObject`s; the landscape is procedually textured based on the steepness of the land, and grass is generated on the surface via a geometry shader. Grass only generates with a certain distance of the viewer.



This landscape is procedurally generated using Perlin noise. The texture used depends on the steepness of the land. When the land is especially steep, a rocky texture is used. Grass blades are generated on flatter land, and only within a certain distance of the user. All the textures used in the image above can found in the `textures/landscape` *folder. All textures come from* `http://www.cgtextures.com/`

`class AABBObject :: public RenderableObject`

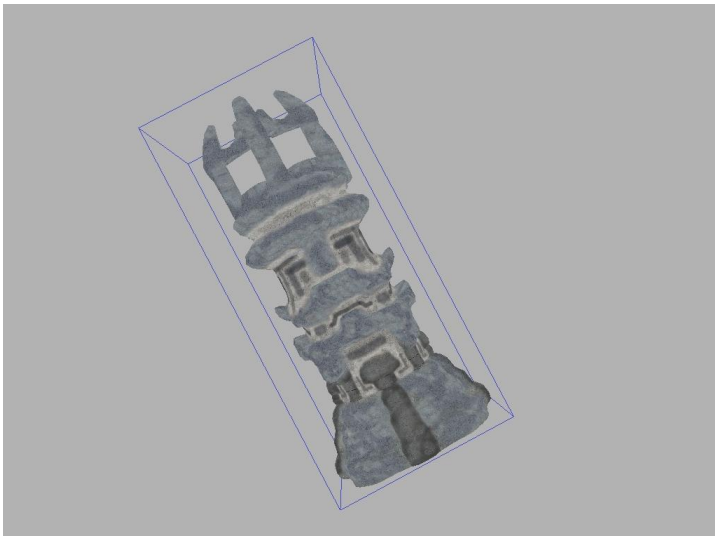
An Axis-Aligned Bounding Box (AABB) is a box that completely contains some object. The edges of an AABB are aligned with x, y, and z axes. In this program, they are used for the puposes of ray casting; a user can click on some location in the window in order to select an object in the scene. If the user clicks on an AABB, the object that the AABB contains is selected as the current object. AABB's are represented as `AABBObject`s in this program.



The AABB in the image above is represented by the red outlined box. When the tower is rotated, the dimensions of the AABB are modified so that the box still contains the tower. Note that the edges of the box remain aligned with the x,y, and z axes.

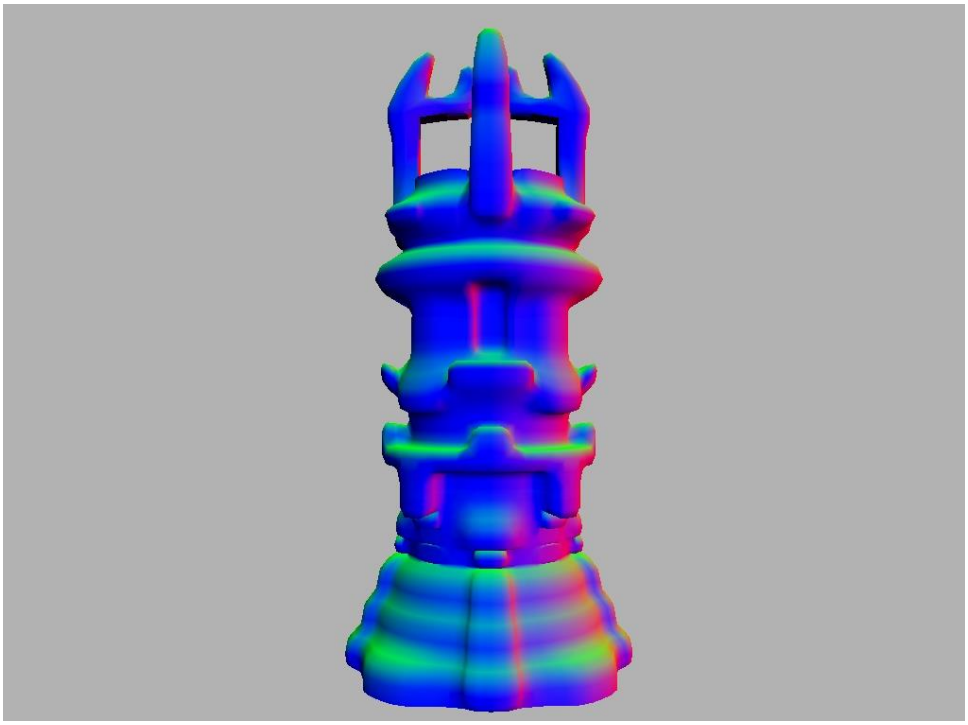
`class OBBObject :: public RenderableObject`

An Oriented Bounding Box (OBB) completely contains some object, much like AABB's. Unlike AABB's, however, the edges of the OBB are aligned with the orientation of the object; if the object is rotated then the OBB is similarly rotated. OBB's are represented as `OBBObject`s in this program. They are currently not used in mouse picking.

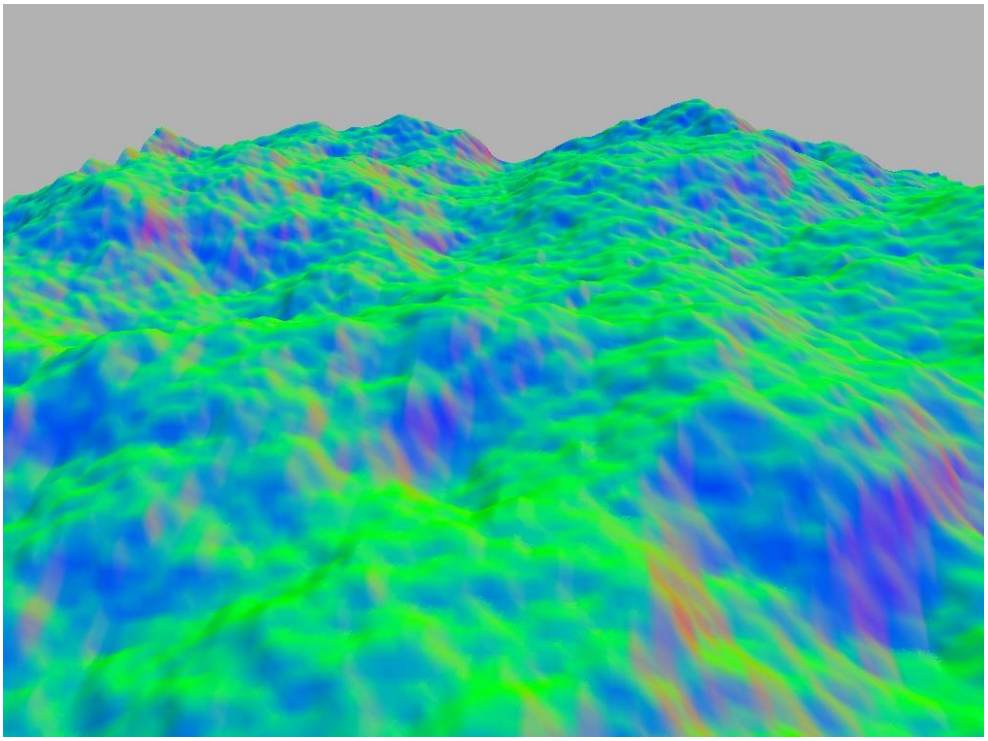


The OBB in the image above is represented by the blue outlined box. When the tower is rotated, the box rotates as well.

5 Other Images



The normals of an `OBJObject`. The normals for the object were directly read from a Wavefront OBJ file.



The normals of a `PerlinHeightmapObject`. Because there is no normal data is directly available for `HeightmapObjects` and `PerlinHeightmapObjects`, they must be computed at the time that the object is created.

A tower being illuminated by a point light source coming from the right.

6 Credits

All sources are also cited in the source code.

(`Keyhandler.h, Mousehandler.h`) Mouse-look code and keyboard movement code adapted from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-6-keyboard-and-mouse/

(`InitGL.h`) Some shader initialization code adapted from http://www.arcsynthesis.org/gltut/Basics/Tuto1%20Making%20Shaders.html

An all-around very helpful website for learning OpenGL. Used this website to implement the Phong reflection model and test the normals of an object in the fragment shaders.
http://antongerdelan.net/opengl/
(`MouseHandler.h`) And for ray-casting code:
http://antongerdelan.net/opengl/raycasting.html

(`AABBObject.`cpp) For ray-AABB intersection tests, used code by zacharmarz from
http://gamedev.stackexchange.com/questions/18436/most-efficient-aabb-vs-ray-collision-algorithms

(`HeightmapObject.`cpp) To generate `HeightmapObject`s, used some code from
http://www.learnopengles.com/android-lesson-eight-an-introduction-to-index-buffer-objects-ibos/

(`Util.cpp`) For fast file reading code:
http://insanecoding.blogspot.com/2011/11/how-to-read-in-file-in-c.html

(see the `heightmaps` and `textures` folders) Used an example heightmap image and texture from
http://www.chadvernon.com/blog/resources/directx9/terrain-generation-with-a-heightmap/