OpenGLTest

Shaneel Sharan

Last Updated: September 3, 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)

2 Using the Program

When the program is initialized, two windows will open. One is the window that OpenGL renders to, and the other window is a console window that displays useful info like the currently selected object.

The user starts at the origin of the world, above a procedurally-generated landscape that is 256 x 256 vertices in dimension. **Mouselook** is enabled, meaning that the user can look around the scene by moving the mouse. To do this, however, the program must take control of the mouse cursor and force it to the center of the rendering window. To regain control of the mouse and disable mouselook, the user can press the M key. A **free-flying camera** is also enabled, so the user can move around freely. Pressing the Y key disables the free-flying camera, causing the user to walk/run on the land they started on. See the controls below for more details.

The controls below show how to use the program and also illuminate other features of this program.

2. 1 Mouse Controls

Moving the mouse: If mouselook is enabled, then this allows the user to look around the scene. If mouselook is disabled, then this simply moves the mouse cursor. Note that the mouse cursor is not visible

when mouselook is enabled (this is done so that the user is not distracted by the mouse cursor at the center of the window).

Left-click: Selects the object that the mouse cursor points to. When mouselook mode is enabled, the mouse cursor is always in the center of the rendering window, even though it isn't visible to the user. In this case, left clicking will result in selecting the object in the center of the screen. The selection process works as follows: when the mouse is clicked, a "ray" is fired into the screen. If this ray collides with the bounding box of an object (which is an approximation of the object), then that object is selected as the current object. To be more specific, each box has two bounding boxes: an Axis-Aligned Bounding Box (AABB; the red outlined box) and an Oriented Bounding Box (OBB; the blue outlined box). The program currently checks for intersections with AABBs only. Note that bounding boxes themselves cannot be directly selected, nor can the landscape that the user starts in.

2.2 Keyboard Controls

ESC: Closes the window and terminates the program.

W, A, S, D: Moves the user around the scene. The type of movement depends on whether or not the free-flying camera is enabled.

M: Toggles mouselook mode on/off.

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 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 on the landscape.

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. This object is added to the current `Scene` object.

2: Generates a tower object (an `OBJObject`) at the user's current location. This object is added to the current `Scene` object.

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. This object is added to the current `Scene` object.

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. This object is added to the current `Scene` object and is selectable by the user as a result. Note that even though the landscape generated at the start of the program is also a `PerlinHeightmapObject`, it is not part of the `Scene` object and therefore not selectable.

3 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).

4 The Main and Render Functions

The `main()` function (in `Main.cpp`) 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) {

    /* Initialization code. */
    initialize SDL
    initialize GLEW
    enable depth buffering, blending, and other OpenGL features as needed
    create the OpenGL shader and program objects
    set up the sampler variables in GLSL
    set up perspective, model, view, and normal matrices (update uniforms in GLSL)

    /* Set up the Scene, object, and lights. */
    create a Scene object that will hold all selectable objects created at runtime
    set up the "landscape" object which user can walk on; it is not added to Scene
    create the point lights

    /* Set up mouse and enter the main loop. */
    initialize the mouse cursor
    while (program should still run) {
        handle keyboard input
        handle mouse input
        render the Scene
    }

    /* Cleanup and return. */
    delete all objects that were created
    de-initialize SDL
    return 0
}
```

First, the function initializes the dependencies (SDL, GLEW). Depth buffering is enabled so that objects appear to be correctly positioned in space, and blending is enabled so that transparent textures (textures with an alpha channel) work. Program objects are then created from the shader code. The sampler variables in the GLSL code are initialized using these program objects, and then the transformation matrices are initialized.

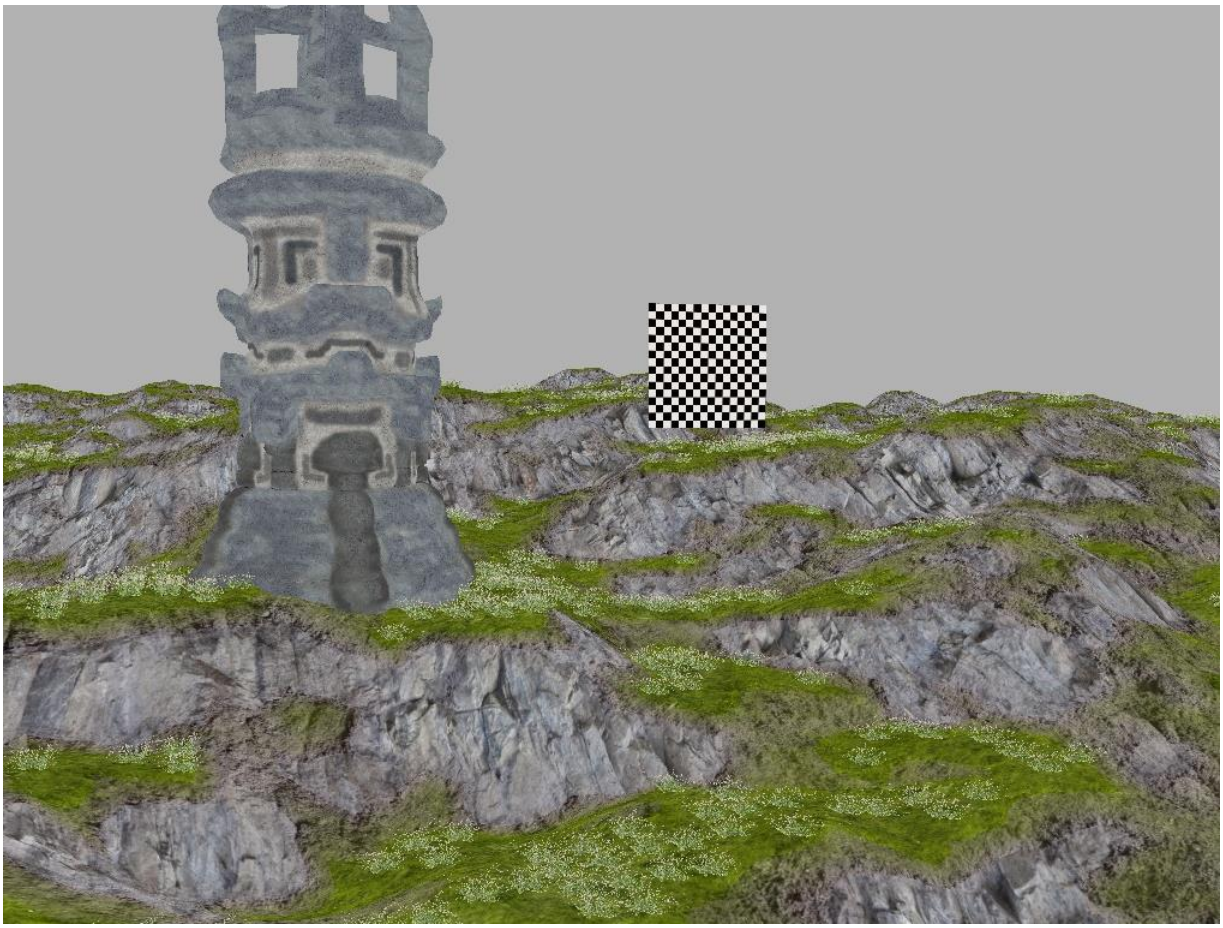
A `Scene` object is then created, which simply contains a list of objects to render and a few utility functions for selecting objects in that list. We can create objects at runtime and add them to the `Scene`.

We then create a `PerlinHeightmapObject` as a landscape on which the user can walk. To tell other parts of the program that this object is a walkable piece of land, we "register" it using the `LandscapeManager` class. This walkable landscape does not become part of the `Scene` object itself because we don't want to be able to select the landscape itself. The reason for this is that the landscape has a large bounding box, so the user will have a difficult time selecting objects close to the landscape. However, it is possible to make a selectable `PerlinHeightmapObject` by pressing the 4 key while the program is running. A `PerlinHeightmapObject` created via this method is added to the `Scene` object, but cannot be walked on since it is not registered with the `LandscapeManager`.

After creating point lights and initializing the mouse position, we enter the program's main loop. In the main loop, we handle any keyboard and mouse input that the user provides and then call the rendering function. The `render()` function is defined in `Render.cpp` and looks like the following, in pseudocode:

```
void render(Scene& scene) {
	clear color and depth buffers based on information in 'scene'
	render all objects in 'scene'
	render the landscape that was registered with the LandscapeManager
	swap buffers
}
```

When the rending function is called, we provide it the `Scene`, and the render function renders all the objects in that `Scene`. It then renders the `PerlinHeightmapObject` that was registered through the `LandscapeManager`. If no landscape was registered, then only the objects in the `Scene` are registered.



The tower and the checkerboard square a part of single `Scene` object. The landscape object was registered through the `LandscapeManager`. By passing the `Scene` object to the rendering function, we get this image.

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

5 The 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:

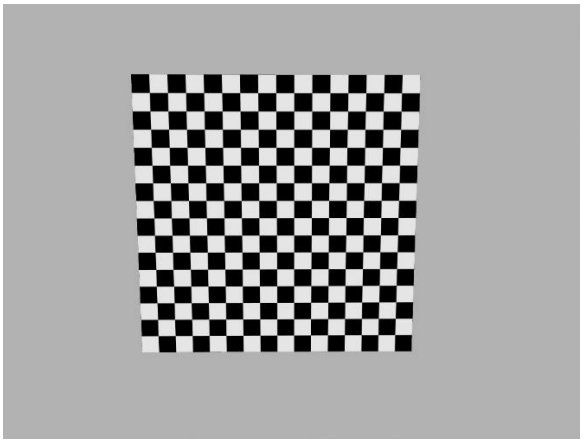
```
RenderableObject
	OBJObject
	HeightmapObject
		PerlinHeightmapObject
	AABBObject
	OBBObject
```

Whenever one of these types of objects is created at runtime, we add it to a `Scene` object, which contains a vector of all the objects to be rendered. We make a `PerlinHeightmapObject` when the program is initialized; this object is not added to the `Scene`. Below is a brief overview of each of the different types of `RenderableObject`s.

5.1 RenderableObject

```
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.

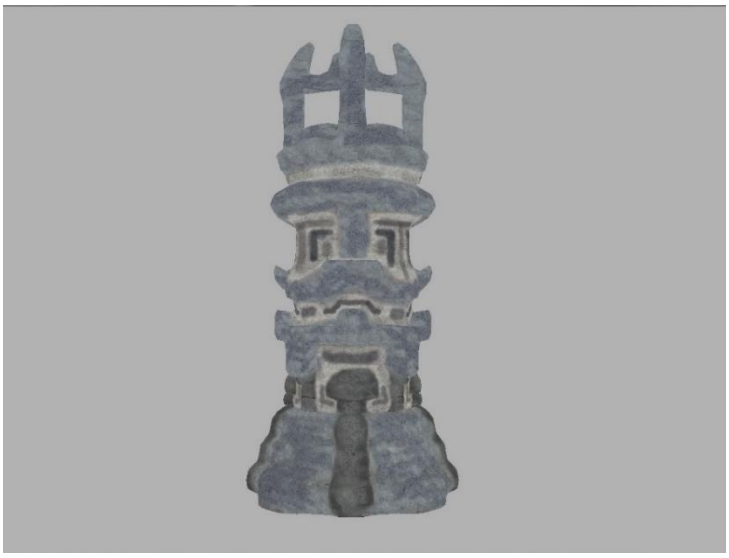
5.2 OBJObject

```
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.

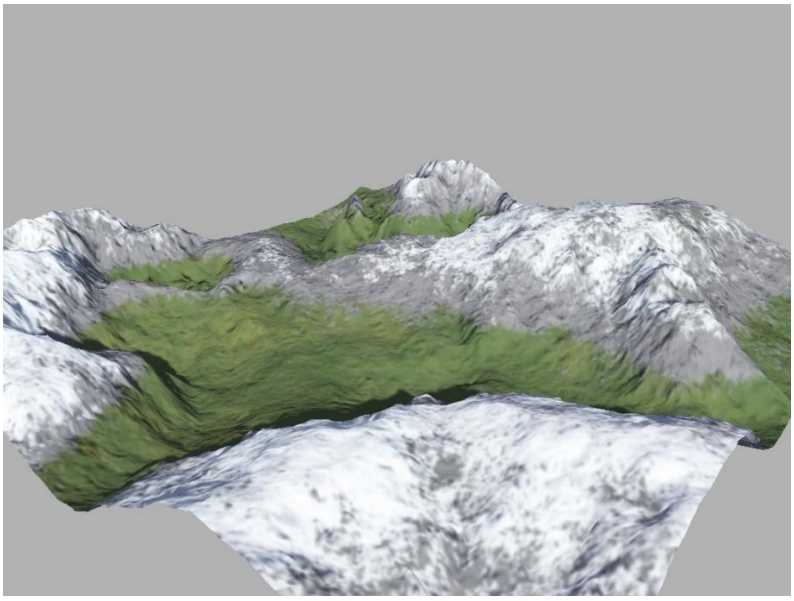


The tower above was created in Blender and then exported as a Wavefront OBJ file. It can found in `obj/tower.obj` *and the associated texture can be found in* `textures/tower.jpg`

5.3 HeightmapObject

```
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).

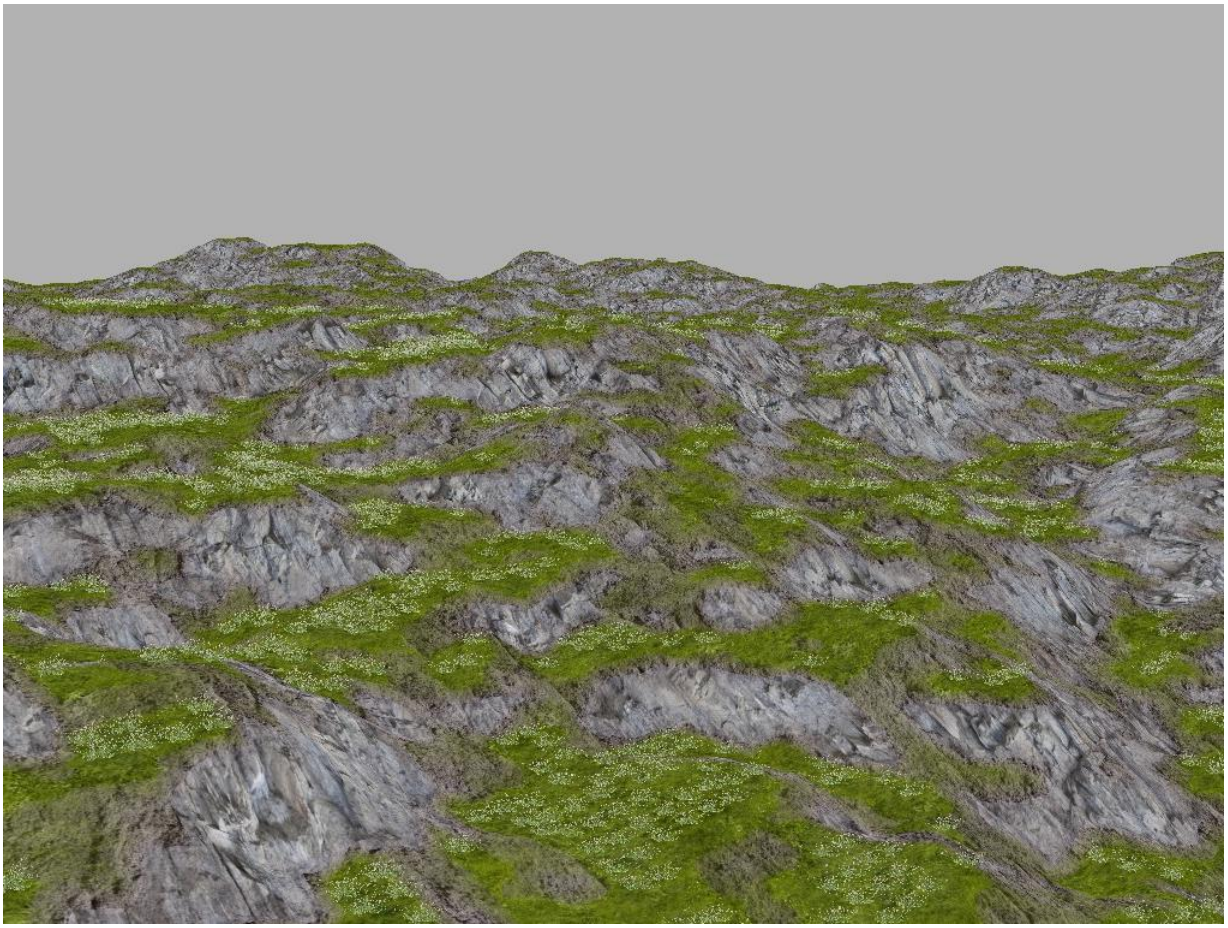


A 256 x 256 `HeightmapObject.` *The heightmap image and the texture used in the image above can be found at* http://www.chadvernon.com/blog/resources/directx9/terrain-generation-with-a-heightmap/

5.4 PerlinHeightmapObject

`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. `PerlinHeightmapObjects` can be used as "landscapes" in the program, so the user can walk on them. Specialized shaders are used for `PerlinHeightmapObjects`; 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.

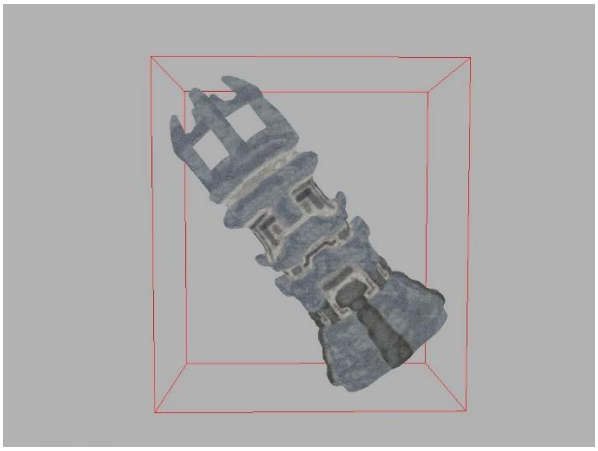


This landscape is procedurally generated using Perlin noise. The texture used depends on the steepness of the land. When the land is especially steep, a rocky texture is used. Grass blades are generated on flatter land, and only within a certain distance of the user. All the textures used in the image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

5.5 AABBObject

```
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.

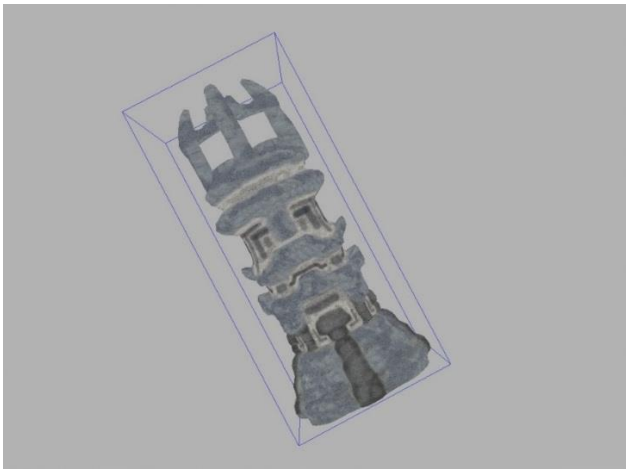


The AABB in the image above is represented by the red outlined box. When the tower is rotated, the dimensions of the AABB are modified so that the box still contains the tower. Note that the edges of the box remain aligned with the x,y, and z axes.

5.6 OBBObject

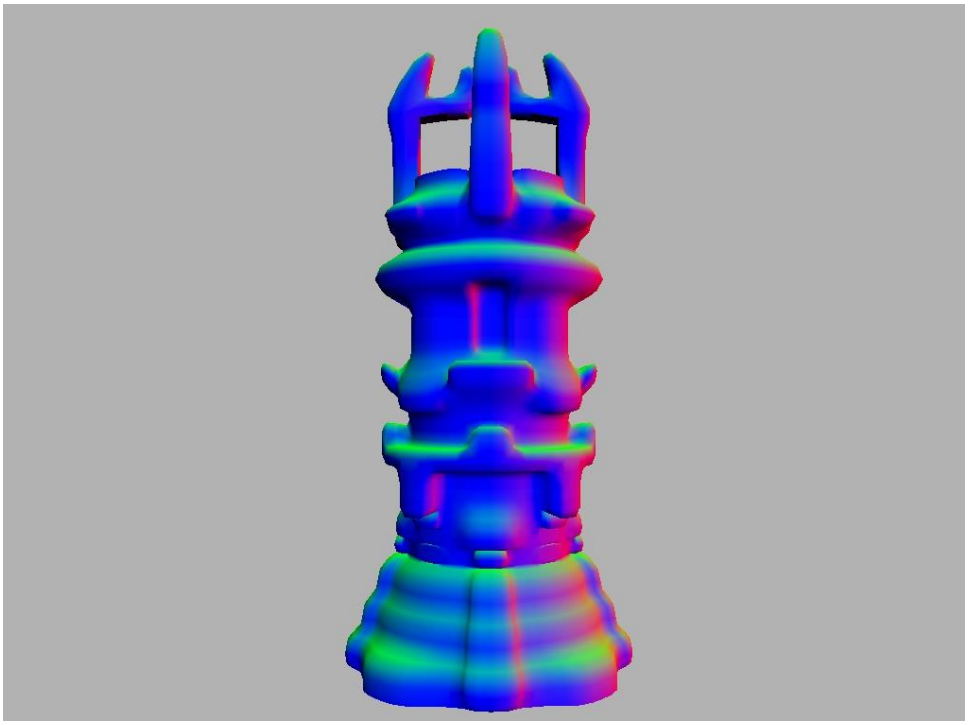
```
class OBBObject :: public RenderableObject
```

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.

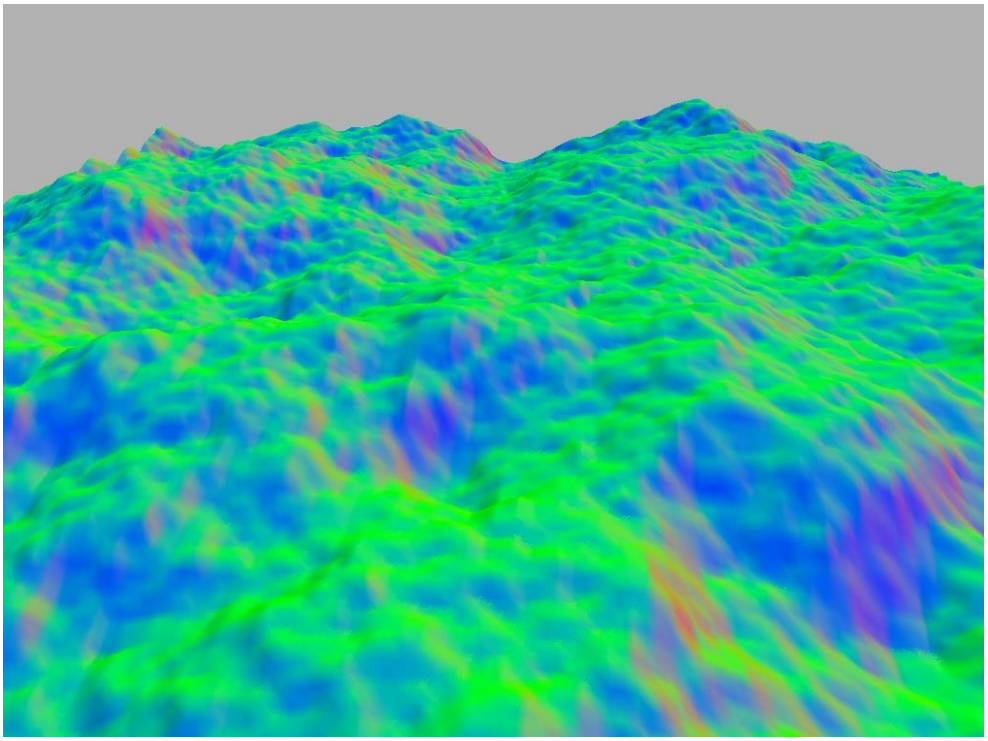


The OBB in the image above is represented by the blue outlined box. When the tower is rotated, the box rotates as well.

6 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.

7 Credits

(`landscape.geo`) An algorithm on generating grass blades
http://http.developer.nvidia.com/GPUGems/gpugems_ch07.html

(`Keyhandler.h, Mousehandler.h`) Mouselook 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/Tut01%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/