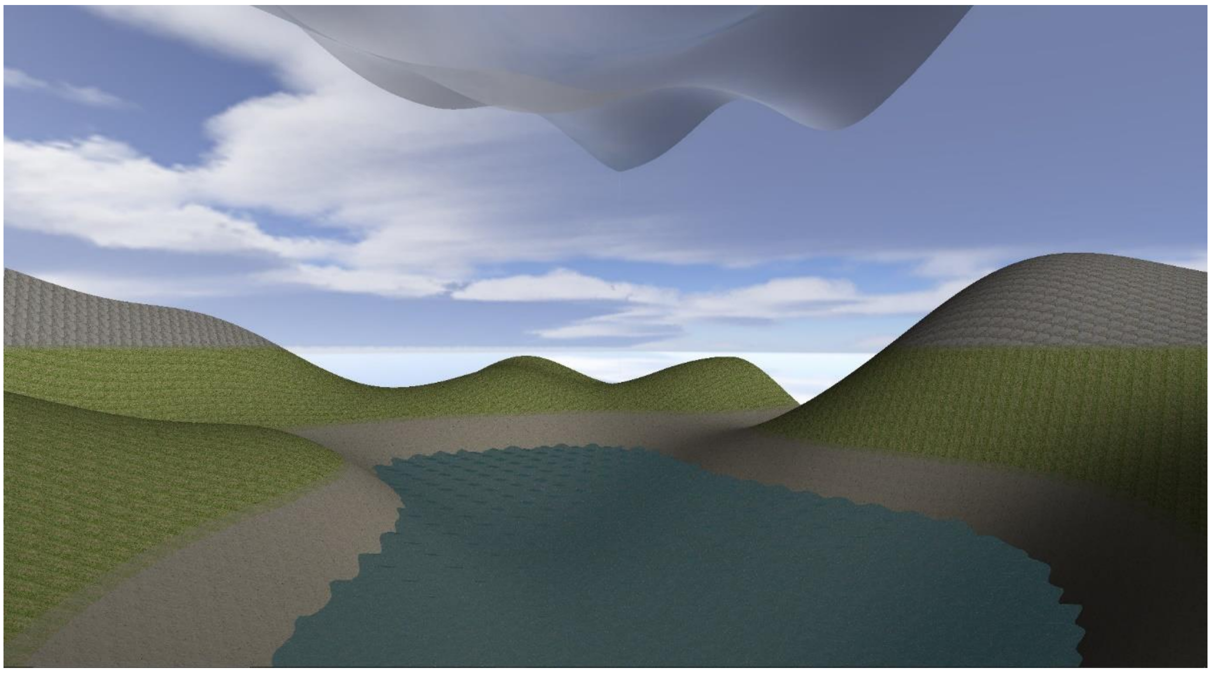
Contents

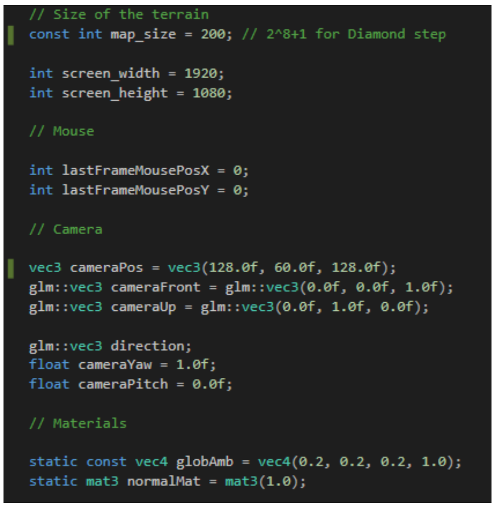
Introduction

This is my procedural terrain generator in OpenGL, C++. It generates a terrain using either the Diamond-Step (Square-step) algorithm, or Perlin noise. This includes a terrain object, cloud object, and water object. These can all have either algorithm used to generate their vertex data, where the final uploaded solution used Perlin for the clouds and terrain and has Diamond-Step commented out for the water. It includes 3D camera controls controlled with WASD and the mouse. The terrain can be regenerated at any time using the R key, and the water and cloud have their own separate animations. All of the objects in the scene have diffuse lighting, and the terrain has multiple textures which are blended depending on the height of the terrain.



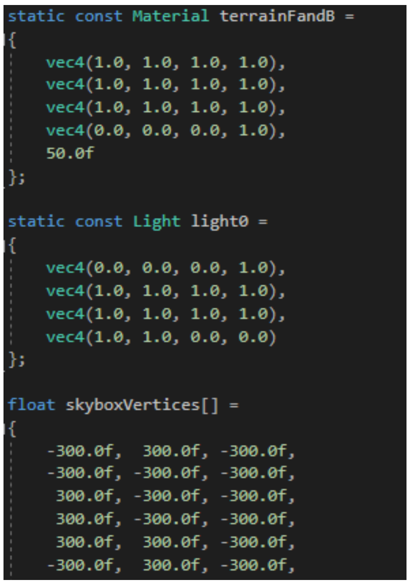
Method

First, I initialize my variables.



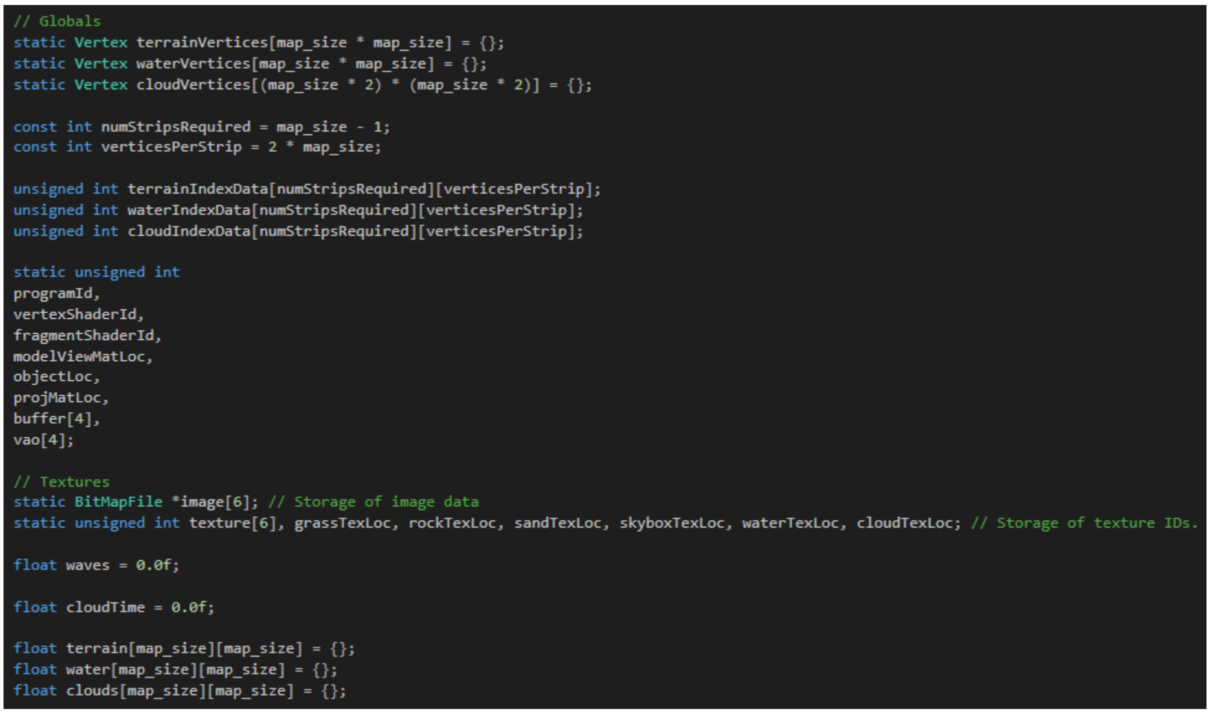
```
// Size of the terrain
const int map_size = 200; // 2^8+1 for Diamond step

int screen_width = 1920;
int screen_height = 1080;

// Mouse

int lastFrameMousePosX = 0;
int lastFrameMousePosY = 0;

// Camera

vec3 cameraPos = vec3(128.0f, 60.0f, 128.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, 1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

glm::vec3 direction;
float cameraYaw = 1.0f;
float cameraPitch = 0.0f;

// Materials

static const vec4 globAmb = vec4(0.2, 0.2, 0.2, 1.0);
static mat3 normalMat = mat3(1.0);
```

- 'map_size' determines the size of all terrain objects. For diamond step, this must be 2 ^ n + 1.
- The camera position and front is used for the model view matrix and projection matrix. The rest of the camera variables are used for the camera movement.
- 'globAmb' is the global light in the scene.
- 'normalMat' is a matrix of normals.



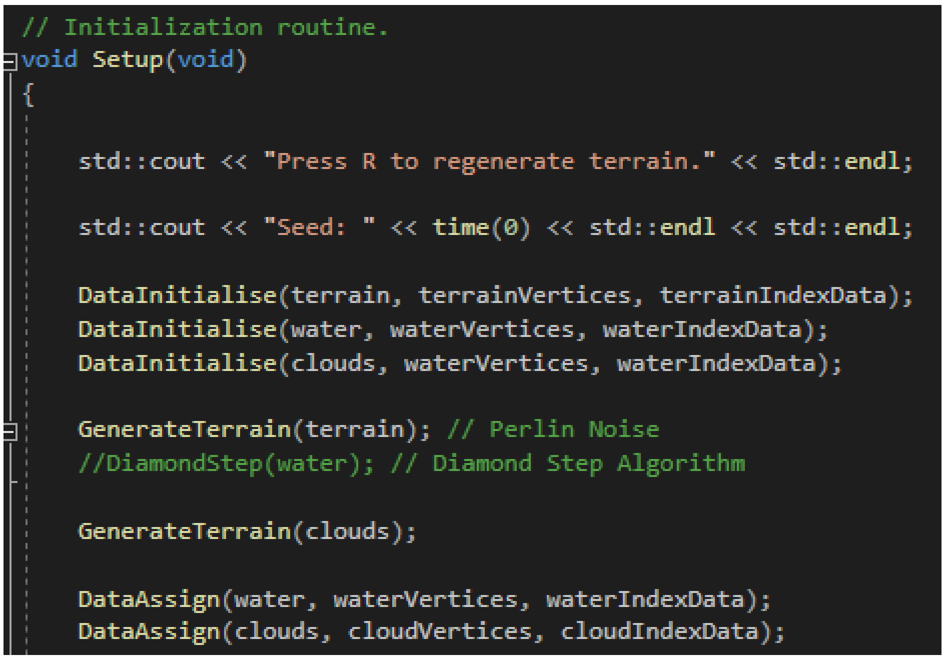
```
static const Material terrainFandB =
{
    vec4(1.0, 1.0, 1.0, 1.0),
    vec4(1.0, 1.0, 1.0, 1.0),
    vec4(1.0, 1.0, 1.0, 1.0),
    vec4(0.0, 0.0, 0.0, 1.0),
    50.0f
};

static const Light light0 =
{
    vec4(0.0, 0.0, 0.0, 1.0),
    vec4(1.0, 1.0, 1.0, 1.0),
    vec4(1.0, 1.0, 1.0, 1.0),
    vec4(1.0, 1.0, 0.0, 0.0)
};

float skyboxVertices[] =
{
    -300.0f,  300.0f, -300.0f,
    -300.0f, -300.0f, -300.0f,
     300.0f, -300.0f, -300.0f,
     300.0f, -300.0f, -300.0f,
     300.0f,  300.0f, -300.0f,
    -300.0f,  300.0f, -300.0f,
```

- ‘terrainFandB’ is the material for the terrain, holding its lighting settings.
- ‘light0’ is the light source in the scene.
- ‘skyboxVertices’ holds all of the vertices for the skybox cubemap.



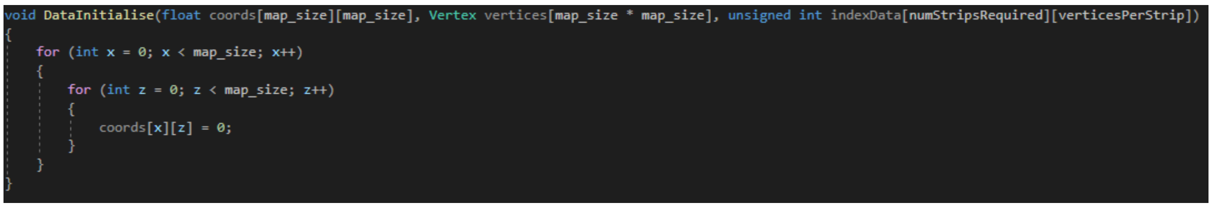
```
// Globals
static Vertex terrainVertices[map_size * map_size] = {};
static Vertex waterVertices[map_size * map_size] = {};
static Vertex cloudVertices[(map_size * 2) * (map_size * 2)] = {};

const int numStripsRequired = map_size - 1;
const int verticesPerStrip = 2 * map_size;

unsigned int terrainIndexData[numStripsRequired][verticesPerStrip];
unsigned int waterIndexData[numStripsRequired][verticesPerStrip];
unsigned int cloudIndexData[numStripsRequired][verticesPerStrip];

static unsigned int
programId,
vertexShaderId,
fragmentShaderId,
modelViewMatLoc,
objectLoc,
projMatLoc,
buffer[4],
vao[4];

// Textures
static BitMapFile *image[6]; // Storage of image data
static unsigned int texture[6], grassTexLoc, rockTexLoc, sandTexLoc, skyboxTexLoc, waterTexLoc, cloudTexLoc; // Storage of texture IDs.

float waves = 0.0f;

float cloudTime = 0.0f;

float terrain[map_size][map_size] = {};
float water[map_size][map_size] = {};
float clouds[map_size][map_size] = {};
```

- ‘terrainVertices’, ‘waterVertices’ and ‘cloudVertices’ holds the vertices for all of the above objects.
- ‘numStripsRequired’ and ‘VerticesPerStrip’ defines how many strips and vertices are used in the triangle strip used in the creation of the terrain.
- ‘indexdata’ holds the strip information for each object.
- ‘programId’ is my openGL program.
- ‘vertexShaderId’ and ‘fragmentShaderId’ are the respective shaders.
- ‘modelViewMatLoc’, ‘objectLoc’ and ‘projMatLoc’ are used to pass information into the shaders.
- ‘buffer’ and ‘vao’ are my object and object vertices stores.
- ‘image’ holds all of my BMP image data.
- ‘texture’ holds all of my texture data.
- ‘waves’ and ‘cloudTime’ are floats that are updated every refresh, controlling the animations for the waves and clouds.
- ‘terrain’, ‘water’, and ‘clouds’ hold all of the height information for the terrains.



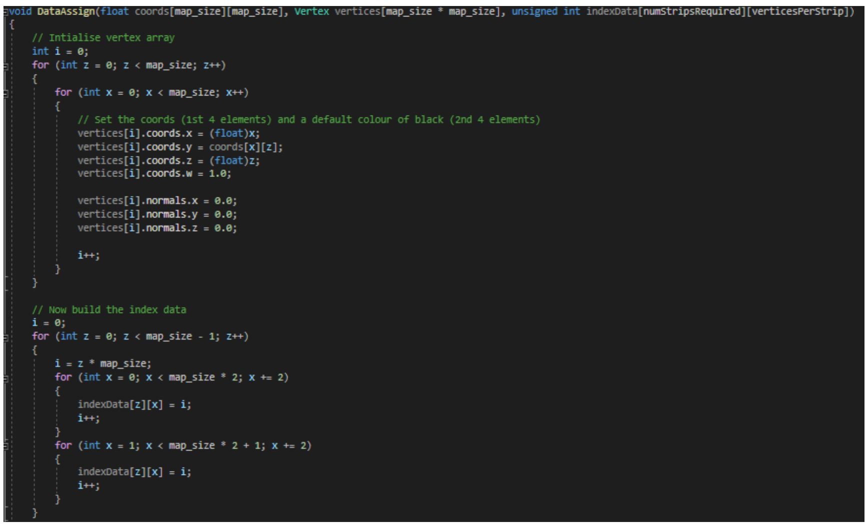
```
// Initialization routine.
void Setup(void)
{

    std::cout << "Press R to regenerate terrain." << std::endl;

    std::cout << "Seed: " << time(0) << std::endl << std::endl;

    DataInitialise(terrain, terrainVertices, terrainIndexData);
    DataInitialise(water, waterVertices, waterIndexData);
    DataInitialise(clouds, waterVertices, waterIndexData);

    GenerateTerrain(terrain); // Perlin Noise
    //DiamondStep(water); // Diamond Step Algorithm

    GenerateTerrain(clouds);

    DataAssign(water, waterVertices, waterIndexData);
    DataAssign(clouds, cloudVertices, cloudIndexData);
```

In my setup function, I first start by initialising all of the terrain data, using the ‘DataInitialise’ and ‘GenerateTerrain’ functions. Then, I assign this data to their respective variables in the DataAssign function.

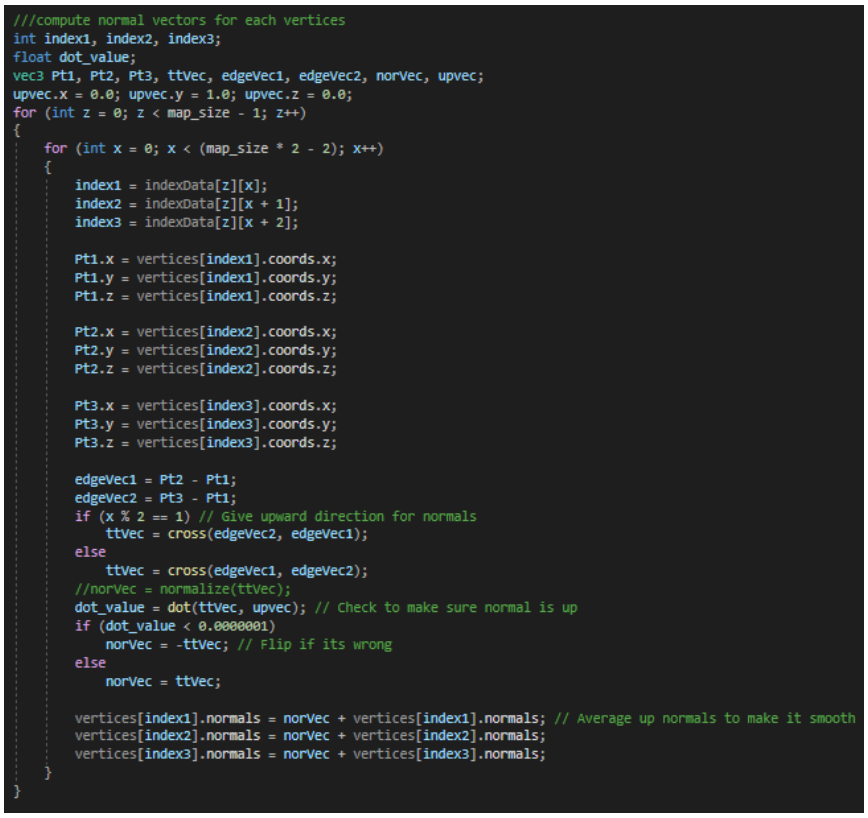


```
void DataInitialise(float coords[map_size][map_size], Vertex vertices[map_size * map_size], unsigned int indexData[numStripsRequired][verticesPerStrip])
{
    for (int x = 0; x < map_size; x++)
    {
        for (int z = 0; z < map_size; z++)
        {
            coords[x][z] = 0;
        }
    }
}
```

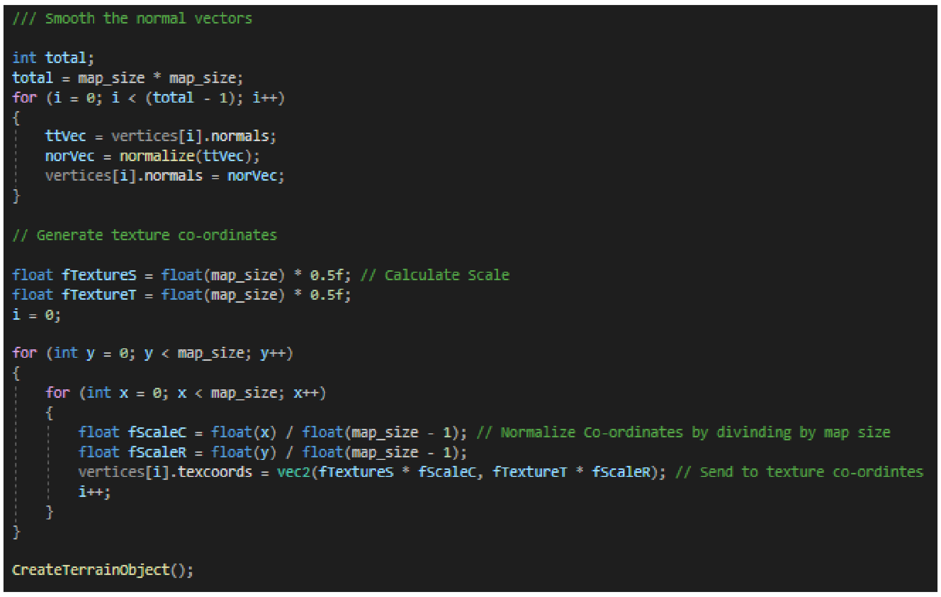
The DataInitialise function initialises the coordinate map for the terrains by setting all of the values to zero.



```
void DataAssign(float coords[map_size][map_size], Vertex vertices[map_size * map_size], unsigned int indexData[numStripsRequired][verticesPerStrip])
{
    // Intialise vertex array
    int i = 0;
    for (int z = 0; z < map_size; z++)
    {
        for (int x = 0; x < map_size; x++)
        {
            // Set the coords (1st 4 elements) and a default colour of black (2nd 4 elements)
            vertices[i].coords.x = (float)x;
            vertices[i].coords.y = coords[x][z];
            vertices[i].coords.z = (float)z;
            vertices[i].coords.w = 1.0;

            vertices[i].normals.x = 0.0;
            vertices[i].normals.y = 0.0;
            vertices[i].normals.z = 0.0;

            i++;
        }
    }

    // Now build the index data
    i = 0;
    for (int z = 0; z < map_size - 1; z++)
    {
        i = z * map_size;
        for (int x = 0; x < map_size * 2; x += 2)
        {
            indexData[z][x] = i;
            i++;
        }
        for (int x = 1; x < map_size * 2 + 1; x += 2)
        {
            indexData[z][x] = i;
            i++;
        }
    }
```

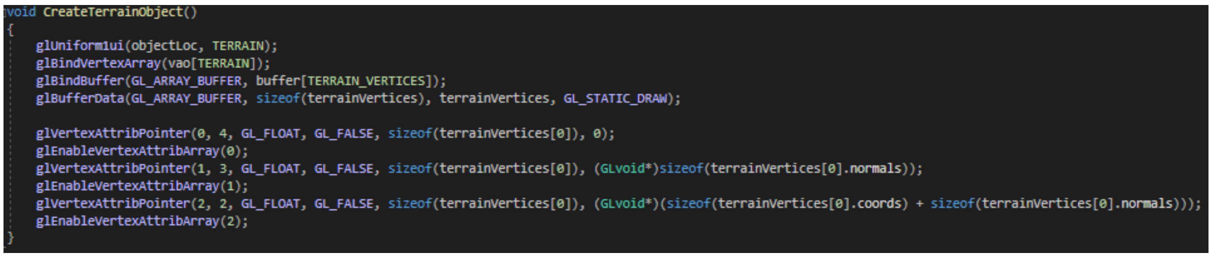


```
///compute normal vectors for each vertices
int index1, index2, index3;
float dot_value;
vec3 Pt1, Pt2, Pt3, ttVec, edgeVec1, edgeVec2, norVec, upvec;
upvec.x = 0.0; upvec.y = 1.0; upvec.z = 0.0;
for (int z = 0; z < map_size - 1; z++)
{
    for (int x = 0; x < (map_size * 2 - 2); x++)
    {
        index1 = indexData[z][x];
        index2 = indexData[z][x + 1];
        index3 = indexData[z][x + 2];

        Pt1.x = vertices[index1].coords.x;
        Pt1.y = vertices[index1].coords.y;
        Pt1.z = vertices[index1].coords.z;

        Pt2.x = vertices[index2].coords.x;
        Pt2.y = vertices[index2].coords.y;
        Pt2.z = vertices[index2].coords.z;

        Pt3.x = vertices[index3].coords.x;
        Pt3.y = vertices[index3].coords.y;
        Pt3.z = vertices[index3].coords.z;

        edgeVec1 = Pt2 - Pt1;
        edgeVec2 = Pt3 - Pt1;
        if (x % 2 == 1) // Give upward direction for normals
            ttVec = cross(edgeVec2, edgeVec1);
        else
            ttVec = cross(edgeVec1, edgeVec2);
        //norVec = normalize(ttVec);
        dot_value = dot(ttVec, upvec); // Check to make sure normal is up
        if (dot_value < 0.0000001)
            norVec = -ttVec; // Flip if its wrong
        else
            norVec = ttVec;

        vertices[index1].normals = norVec + vertices[index1].normals; // Average up normals to make it smooth
        vertices[index2].normals = norVec + vertices[index2].normals;
        vertices[index3].normals = norVec + vertices[index3].normals;
    }
}
```



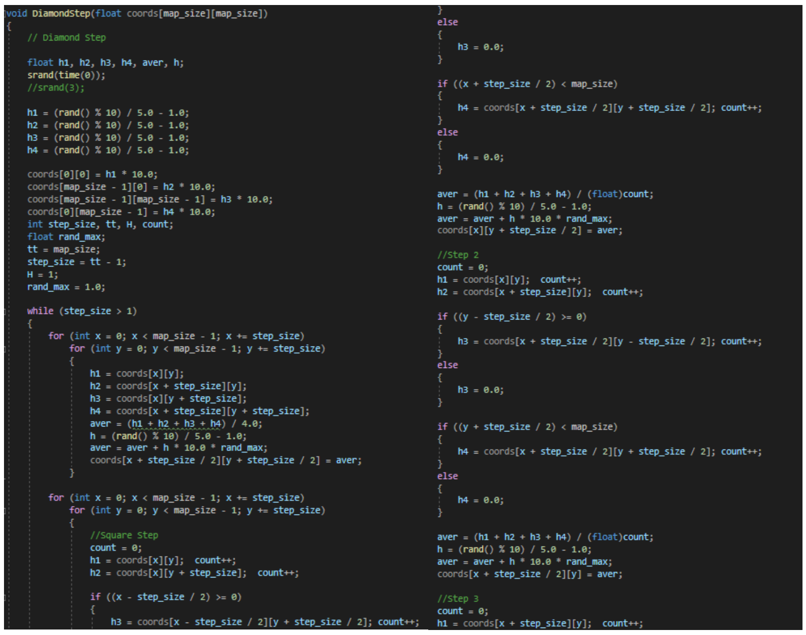
```
/// Smooth the normal vectors

int total;
total = map_size * map_size;
for (i = 0; i < (total - 1); i++)
{
    ttVec = vertices[i].normals;
    norVec = normalize(ttVec);
    vertices[i].normals = norVec;
}

// Generate texture co-ordinates

float fTextureS = float(map_size) * 0.5f; // Calculate Scale
float fTextureT = float(map_size) * 0.5f;
i = 0;

for (int y = 0; y < map_size; y++)
{
    for (int x = 0; x < map_size; x++)
    {
        float fScaleC = float(x) / float(map_size - 1); // Normalize Co-ordinates by dividing by map size
        float fScaleR = float(y) / float(map_size - 1);
        vertices[i].texcoords = vec2(fTextureS * fScaleC, fTextureT * fScaleR); // Send to texture co-ordintes
        i++;
    }
}

CreateTerrainObject();
```

The DataAssign function begins by initialising the vertex array, settings all coordinates to their respective values from the terrain map and the normals to zero. Then, it builds the index data for the terrain object, settings each value to based on ‘I = z * map_size’. Next, it computes the normal vectors for all vertices in the terrain. Using the index data that was just built, it goes through every set of triangles and assigns a coordinate value for each of them. Then, it calculates each normal as well, giving it an upward direction and running a dot check to make sure the normal is pointing up before flipping it if it is not. These normals are averaged up before being normalized in a for loop to make them smooth. Finally, the function generates texture co-ordinates as well, which are given a scale and divided by the map size to normalize them. Finally, it runs the ‘CreateTerrainObject’, which is used to bind the vertices to the shaders in the event that the terrain generation is refreshed.



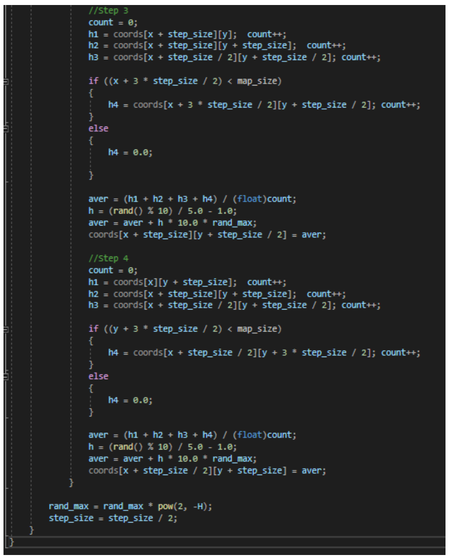
```
void CreateTerrainObject()
{
    glUniform1ui(objectLoc, TERRAIN);
    glBindVertexArray(vao[TERRAIN]);
    glBindBuffer(GL_ARRAY_BUFFER, buffer[TERRAIN_VERTICES]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(terrainVertices), terrainVertices, GL_STATIC_DRAW);

    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, sizeof(terrainVertices[0]), 0);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(terrainVertices[0]), (GLvoid*)sizeof(terrainVertices[0].normals));
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(terrainVertices[0]), (GLvoid*)(sizeof(terrainVertices[0].coords) + sizeof(terrainVertices[0].normals)));
    glEnableVertexAttribArray(2);
}
```

This function simply binds the vertex array and buffer to the corresponding terrain VAO and VBO, and assigns the ‘terrainVertices’ as well as their normals and coordinates to the vertex shader. These go into location 0, 1 and 2 respectively, which oversee the terrain drawing. Also, the type of object is also passed onto both shaders to specify how the object should be rendered.



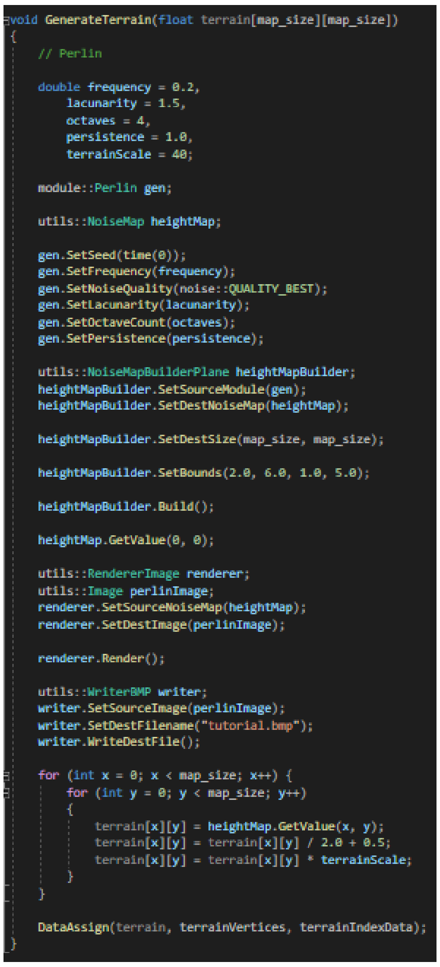
```
void DiamondStep(float coords[map_size][map_size])
{
    // Diamond Step

    float h1, h2, h3, h4, aver, h;
    srand(time(0));
    //srand(3);

    h1 = (rand() % 10) / 5.0 - 1.0;
    h2 = (rand() % 10) / 5.0 - 1.0;
    h3 = (rand() % 10) / 5.0 - 1.0;
    h4 = (rand() % 10) / 5.0 - 1.0;

    coords[0][0] = h1 * 10.0;
    coords[map_size - 1][0] = h2 * 10.0;
    coords[map_size - 1][map_size - 1] = h3 * 10.0;
    coords[0][map_size - 1] = h4 * 10.0;
    int step_size, tt, H, count;
    float rand_max;
    tt = map_size;
    step_size = tt - 1;
    H = 1;
    rand_max = 1.0;

    while (step_size > 1)
    {
        for (int x = 0; x < map_size - 1; x += step_size)
            for (int y = 0; y < map_size - 1; y += step_size)
            {
                h1 = coords[x][y];
                h2 = coords[x + step_size][y];
                h3 = coords[x][y + step_size];
                h4 = coords[x + step_size][y + step_size];
                aver = (h1 + h2 + h3 + h4) / 4.0;
                h = (rand() % 10) / 5.0 - 1.0;
                aver = aver + h * 10.0 * rand_max;
                coords[x + step_size / 2][y + step_size / 2] = aver;
            }

        for (int x = 0; x < map_size - 1; x += step_size)
            for (int y = 0; y < map_size - 1; y += step_size)
            {
                //Square Step
                count = 0;
                h1 = coords[x][y];  count++;
                h2 = coords[x][y + step_size];  count++;

                if ((x - step_size / 2) >= 0)
                {
                    h3 = coords[x - step_size / 2][y + step_size / 2]; count++;
```

```
}
else
{
    h3 = 0.0;
}

if ((x + step_size / 2) < map_size)
{
    h4 = coords[x + step_size / 2][y + step_size / 2]; count++;
}
else
{
    h4 = 0.0;
}

aver = (h1 + h2 + h3 + h4) / (float)count;
h = (rand() % 10) / 5.0 - 1.0;
aver = aver + h * 10.0 * rand_max;
coords[x][y + step_size / 2] = aver;

//Step 2
count = 0;
h1 = coords[x][y];  count++;
h2 = coords[x + step_size][y];  count++;

if ((y - step_size / 2) >= 0)
{
    h3 = coords[x + step_size / 2][y - step_size / 2]; count++;
}
else
{
    h3 = 0.0;
}

if ((y + step_size / 2) < map_size)
{
    h4 = coords[x + step_size / 2][y + step_size / 2]; count++;
}
else
{
    h4 = 0.0;
}

aver = (h1 + h2 + h3 + h4) / (float)count;
h = (rand() % 10) / 5.0 - 1.0;
aver = aver + h * 10.0 * rand_max;
coords[x + step_size / 2][y] = aver;

//Step 3
count = 0;
h1 = coords[x + step_size][y];  count++;
```



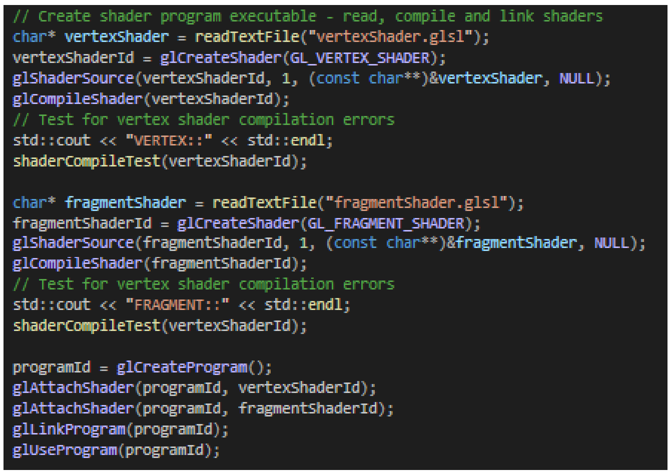
```
                //Step 3
                count = 0;
                h1 = coords[x + step_size][y];  count++;
                h2 = coords[x + step_size][y + step_size];  count++;
                h3 = coords[x + step_size / 2][y + step_size / 2]; count++;

                if ((x + 3 * step_size / 2) < map_size)
                {
                    h4 = coords[x + 3 * step_size / 2][y + step_size / 2]; count++;
                }
                else
                {
                    h4 = 0.0;

                }

                aver = (h1 + h2 + h3 + h4) / (float)count;
                h = (rand() % 10) / 5.0 - 1.0;
                aver = aver + h * 10.0 * rand_max;
                coords[x + step_size][y + step_size / 2] = aver;

                //Step 4
                count = 0;
                h1 = coords[x][y + step_size];  count++;
                h2 = coords[x + step_size][y + step_size];  count++;
                h3 = coords[x + step_size / 2][y + step_size / 2]; count++;

                if ((y + 3 * step_size / 2) < map_size)
                {
                    h4 = coords[x + step_size / 2][y + 3 * step_size / 2]; count++;
                }
                else
                {
                    h4 = 0.0;
                }

                aver = (h1 + h2 + h3 + h4) / (float)count;
                h = (rand() % 10) / 5.0 - 1.0;
                aver = aver + h * 10.0 * rand_max;
                coords[x + step_size / 2][y + step_size] = aver;
            }

        rand_max = rand_max * pow(2, -H);
        step_size = step_size / 2;
    }
}
```

The diamond step function applies the diamond-square algorithm to the object map. It takes it's coordinates as input, and transforms them based on a random range which is seeded using the current time. This function takes the object coordinate array, which should have a height of "2n + 1", and sets its for corners as the initial values. Then, it sets the midpoint of each square to be the average result of the four corner points added to a random value. This generates a 'diamond' pattern. It then takes each of these diamonds and sets their midpoint in the same way, with a random value added on. It checks to make sure there are four points, as the edges will have only three and cannot be used in which case it simply sets it to zero. With every step, more and more midpoints are calculated, creating more diamonds with random heights, therefore generating a height map for the terrain.



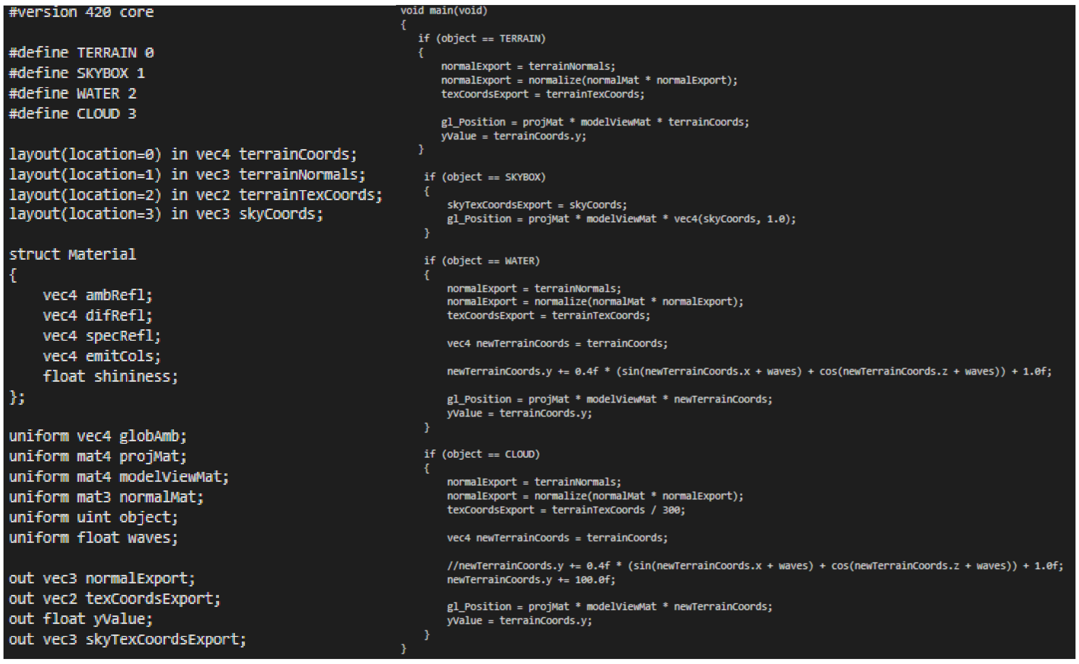
```
void GenerateTerrain(float terrain[map_size][map_size])
{
    // Perlin

    double frequency = 0.2,
        lacunarity = 1.5,
        octaves = 4,
        persistence = 1.0,
        terrainScale = 40;

    module::Perlin gen;

    utils::NoiseMap heightMap;

    gen.SetSeed(time(0));
    gen.SetFrequency(frequency);
    gen.SetNoiseQuality(noise::QUALITY_BEST);
    gen.SetLacunarity(lacunarity);
    gen.SetOctaveCount(octaves);
    gen.SetPersistence(persistence);

    utils::NoiseMapBuilderPlane heightMapBuilder;
    heightMapBuilder.SetSourceModule(gen);
    heightMapBuilder.SetDestNoiseMap(heightMap);

    heightMapBuilder.SetDestSize(map_size, map_size);

    heightMapBuilder.SetBounds(2.0, 6.0, 1.0, 5.0);

    heightMapBuilder.Build();

    heightMap.GetValue(0, 0);

    utils::RendererImage renderer;
    utils::Image perlinImage;
    renderer.SetSourceNoiseMap(heightMap);
    renderer.SetDestImage(perlinImage);

    renderer.Render();

    utils::WriterBMP writer;
    writer.SetSourceImage(perlinImage);
    writer.SetDestFilename("tutorial.bmp");
    writer.WriteDestFile();

    for (int x = 0; x < map_size; x++) {
        for (int y = 0; y < map_size; y++)
        {
            terrain[x][y] = heightMap.GetValue(x, y);
            terrain[x][y] = terrain[x][y] / 2.0 + 0.5;
            terrain[x][y] = terrain[x][y] * terrainScale;
        }
    }

    DataAssign(terrain, terrainVertices, terrainIndexData);
}
```

The generate terrain function is the second height map function I have created and is used for the terrain in the final build. This function uses Perlin noise instead of the Diamond-Step algorithm, giving much smoother and more realistic results. This is using the 'libnoise' library to generate a new heightmap image each time. I have set my own values for the frequency, lacunarity, octaves, persistence and scale of the Perlin noise in order to create a result that I am happy with, and these values could be randomly generated or manually input to generate much different looking terrains. Libnoise takes these variables and applies them to a heightmap, of which values I place into the terrain coordinate map for each coordinate. I divide this by 2 and add 0.5 as the heightmap returns a value between -1 and 1, whereas I want a value between 0 and 1. Finally, I once again assign the terrain data with the DataAssign function, as the generation function may be run inside the program.



```cpp
// Create shader program executable - read, compile and link shaders
char* vertexShader = readTextFile("vertexShader.glsl");
vertexShaderId = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShaderId, 1, (const char**)&vertexShader, NULL);
glCompileShader(vertexShaderId);
// Test for vertex shader compilation errors
std::cout << "VERTEX::" << std::endl;
shaderCompileTest(vertexShaderId);

char* fragmentShader = readTextFile("fragmentShader.glsl");
fragmentShaderId = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShaderId, 1, (const char**)&fragmentShader, NULL);
glCompileShader(fragmentShaderId);
// Test for vertex shader compilation errors
std::cout << "FRAGMENT::" << std::endl;
shaderCompileTest(vertexShaderId);

programId = glCreateProgram();
glAttachShader(programId, vertexShaderId);
glAttachShader(programId, fragmentShaderId);
glLinkProgram(programId);
glUseProgram(programId);
```

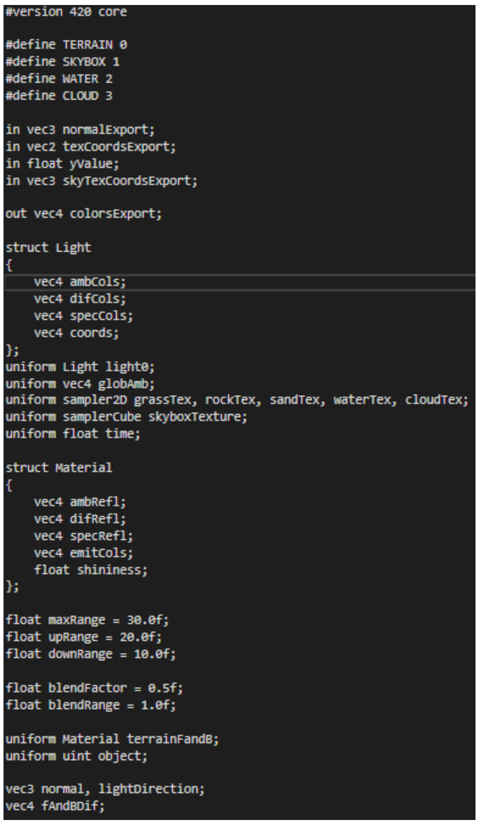
Back in the setup function, I create the shader program executables for the vertex shader and fragment shader, as well as assign the programId. To attach the shaders to the openGL program.



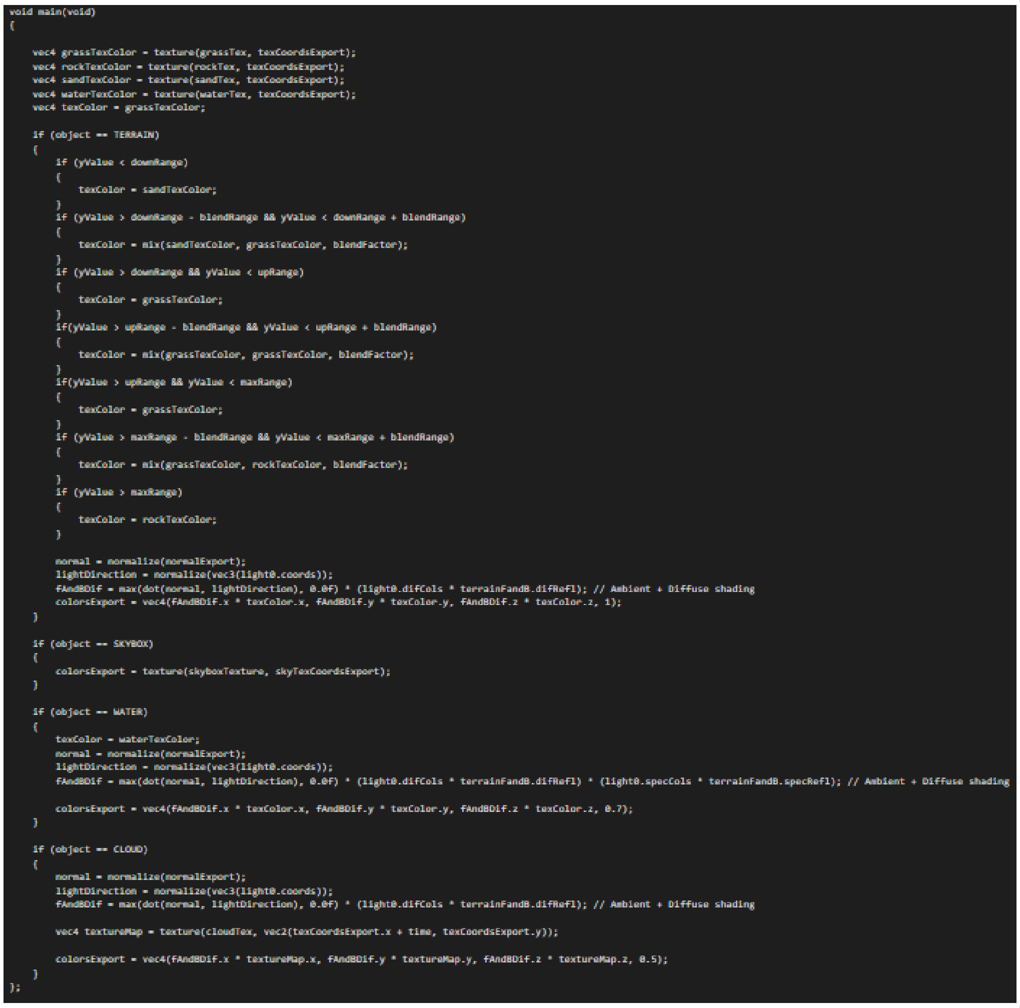
```glsl
#version 420 core

#define TERRAIN 0
#define SKYBOX 1
#define WATER 2
#define CLOUD 3

layout(location=0) in vec4 terrainCoords;
layout(location=1) in vec3 terrainNormals;
layout(location=2) in vec2 terrainTexCoords;
layout(location=3) in vec3 skyCoords;

struct Material
{
    vec4 ambRefl;
    vec4 difRefl;
    vec4 specRefl;
    vec4 emitCols;
    float shininess;
};

uniform vec4 globAmb;
uniform mat4 projMat;
uniform mat4 modelViewMat;
uniform mat3 normalMat;
uniform uint object;
uniform float waves;

out vec3 normalExport;
out vec2 texCoordsExport;
out float yValue;
out vec3 skyTexCoordsExport;
```

```glsl
void main(void)
{
   if (object == TERRAIN)
   {
      normalExport = terrainNormals;
      normalExport = normalize(normalMat * normalExport);
      texCoordsExport = terrainTexCoords;

      gl_Position = projMat * modelViewMat * terrainCoords;
      yValue = terrainCoords.y;
   }

    if (object == SKYBOX)
    {
       skyTexCoordsExport = skyCoords;
       gl_Position = projMat * modelViewMat * vec4(skyCoords, 1.0);
    }

    if (object == WATER)
    {
       normalExport = terrainNormals;
       normalExport = normalize(normalMat * normalExport);
       texCoordsExport = terrainTexCoords;

       vec4 newTerrainCoords = terrainCoords;

       newTerrainCoords.y += 0.4f * (sin(newTerrainCoords.x + waves) + cos(newTerrainCoords.z + waves)) + 1.0f;

       gl_Position = projMat * modelViewMat * newTerrainCoords;
       yValue = terrainCoords.y;
    }

    if (object == CLOUD)
    {
       normalExport = terrainNormals;
       normalExport = normalize(normalMat * normalExport);
       texCoordsExport = terrainTexCoords / 300;

       vec4 newTerrainCoords = terrainCoords;

       //newTerrainCoords.y += 0.4f * (sin(newTerrainCoords.x + waves) + cos(newTerrainCoords.z + waves)) + 1.0f;
       newTerrainCoords.y += 100.0f;

       gl_Position = projMat * modelViewMat * newTerrainCoords;
       yValue = terrainCoords.y;
    }
}
```

The vertex shader is in charge of performing operations on the object’s vertex data, depending on which type of object they are. The layout locations are passed in to the shader from vertex attribute pointers, bringing in information such as their coordinates, normals and texture coordinates. Some variables such as ‘waves’ are also passed in, which are used for the sine animation of the water. Although it depends on the objects, they all pass out a position consisting of their coordinates multiplied by the projection matrix and view matrix, and a texture coordinates vector which is used in the fragment shader to apply the correct texture onto each object. The water object uses the function “y = sine (x + 5) – cosine (z + t)”, where t is the wave variable. This applies a fluctuation property to the water object which is animated as the time variable changes on every refresh.

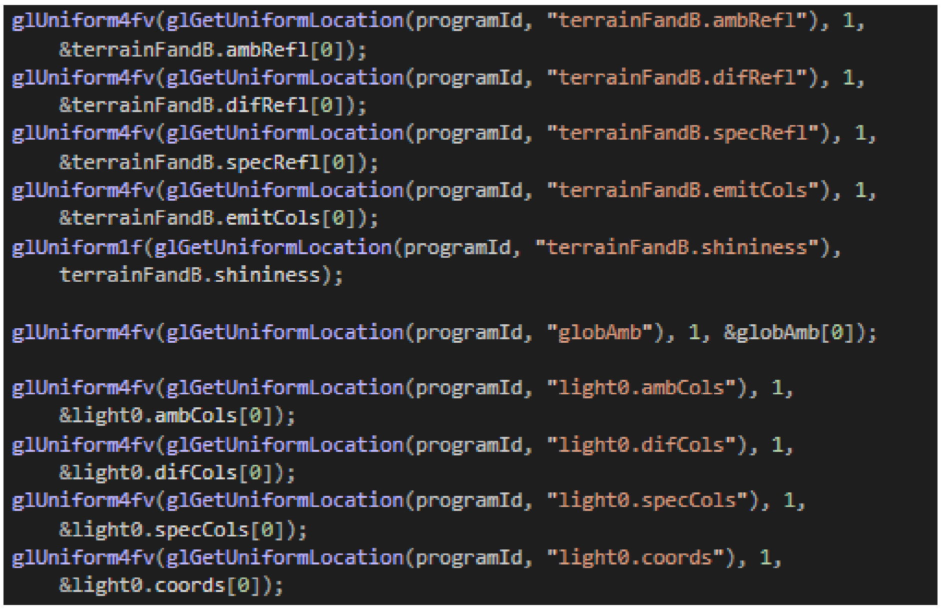


```
#version 420 core

#define TERRAIN 0
#define SKYBOX 1
#define WATER 2
#define CLOUD 3

in vec3 normalExport;
in vec2 texCoordsExport;
in float yValue;
in vec3 skyTexCoordsExport;

out vec4 colorsExport;

struct Light
{
    vec4 ambCols;
    vec4 difCols;
    vec4 specCols;
    vec4 coords;
};
uniform Light light0;
uniform vec4 globAmb;
uniform sampler2D grassTex, rockTex, sandTex, waterTex, cloudTex;
uniform samplerCube skyboxTexture;
uniform float time;

struct Material
{
    vec4 ambRefl;
    vec4 difRefl;
    vec4 specRefl;
    vec4 emitCols;
    float shininess;
};

float maxRange = 30.0f;
float upRange = 20.0f;
float downRange = 10.0f;

float blendFactor = 0.5f;
float blendRange = 1.0f;

uniform Material terrainFandB;
uniform uint object;

vec3 normal, lightDirection;
vec4 fAndBDif;
```



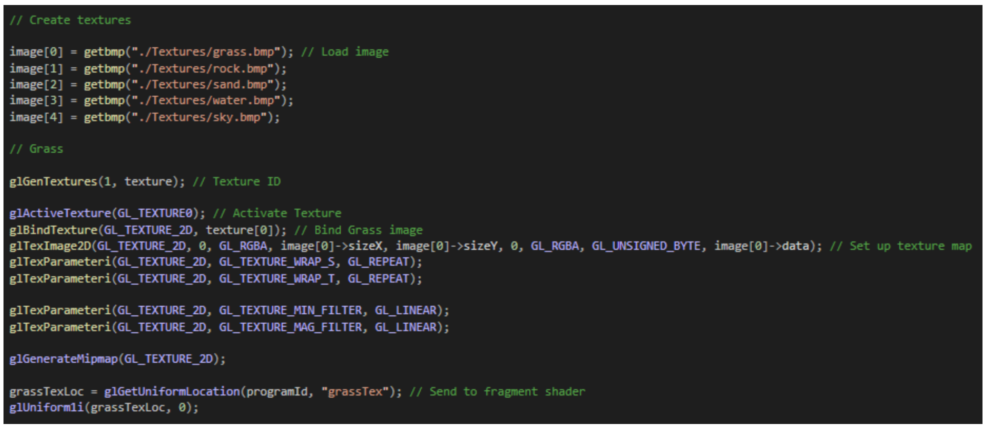
```
void main(void)
{

    vec4 grassTexColor = texture(grassTex, texCoordsExport);
    vec4 rockTexColor = texture(rockTex, texCoordsExport);
    vec4 sandTexColor = texture(sandTex, texCoordsExport);
    vec4 waterTexColor = texture(waterTex, texCoordsExport);
    vec4 texColor = grassTexColor;

    if (object == TERRAIN)
    {
        if (yValue < downRange)
        {
            texColor = sandTexColor;
        }
        if (yValue > downRange - blendRange && yValue < downRange + blendRange)
        {
            texColor = mix(sandTexColor, grassTexColor, blendFactor);
        }
        if (yValue > downRange && yValue < upRange)
        {
            texColor = grassTexColor;
        }
        if(yValue > upRange - blendRange && yValue < upRange + blendRange)
        {
            texColor = mix(grassTexColor, grassTexColor, blendFactor);
        }
        if(yValue > upRange && yValue < maxRange)
        {
            texColor = grassTexColor;
        }
        if (yValue > maxRange - blendRange && yValue < maxRange + blendRange)
        {
            texColor = mix(grassTexColor, rockTexColor, blendFactor);
        }
        if (yValue > maxRange)
        {
            texColor = rockTexColor;
        }

        normal = normalize(normalExport);
        lightDirection = normalize(vec3(light0.coords));
        fAndBDif = max(dot(normal, lightDirection), 0.0f) * (light0.difCols * terrainFandB.difRefl); // Ambient + Diffuse shading
        colorsExport = vec4(fAndBDif.x * texColor.x, fAndBDif.y * texColor.y, fAndBDif.z * texColor.z, 1);
    }

    if (object == SKYBOX)
    {
        colorsExport = texture(skyboxTexture, skyTexCoordsExport);
    }

    if (object == WATER)
    {
        texColor = waterTexColor;
        normal = normalize(normalExport);
        lightDirection = normalize(vec3(light0.coords));
        fAndBDif = max(dot(normal, lightDirection), 0.0f) * (light0.difCols * terrainFandB.difRefl) * (light0.specCols * terrainFandB.specRefl); // Ambient + Diffuse shading

        colorsExport = vec4(fAndBDif.x * texColor.x, fAndBDif.y * texColor.y, fAndBDif.z * texColor.z, 0.7);
    }

    if (object == CLOUD)
    {
        normal = normalize(normalExport);
        lightDirection = normalize(vec3(light0.coords));
        fAndBDif = max(dot(normal, lightDirection), 0.0f) * (light0.difCols * terrainFandB.difRefl); // Ambient + Diffuse shading

        vec4 textureMap = texture(cloudTex, vec2(texCoordsExport.x + time, texCoordsExport.y));

        colorsExport = vec4(fAndBDif.x * textureMap.x, fAndBDif.y * textureMap.y, fAndBDif.z * textureMap.z, 0.5);
    }
};
```

The fragment shader applies materials onto the objects based on their texture coordinates. Once again, it takes in variables such as time and the various textures, as well as the normals and texture coordinates from the vertex shader. Additionally, it takes a ‘yValue’ variable which is used to check the elevation of the vertices and apply a different material depending on what this height is. For the terrain object, there is a max, up and down range, which is a value that decides which material should be applied at that height. In between those, there is a blend range which applies a mixture of the two materials to make the transition between them gradual. Finally, the normals are normalized and used to calculate the colour export of the object, which uses ambient and diffuse shading calculated from ‘light0’, and the terrain material properties. This is exported through a vector4, with the last value being the alpha transparency of the object. For instance, the water and clouds have a value of less than 1 to make them see through.



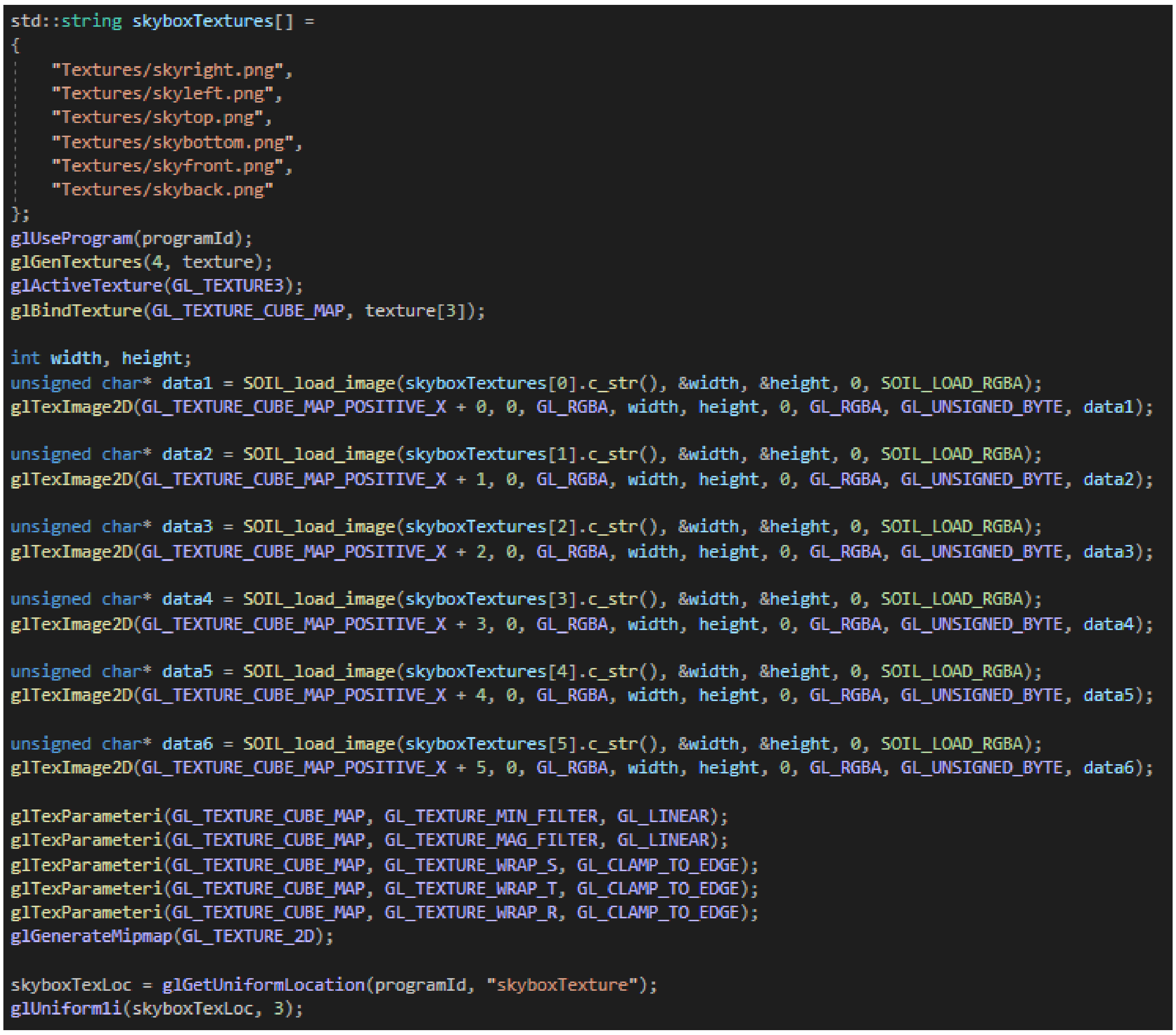
```
glUniform4fv(glGetUniformLocation(programId, "terrainFandB.ambRefl"), 1,
    &terrainFandB.ambRefl[0]);
glUniform4fv(glGetUniformLocation(programId, "terrainFandB.difRefl"), 1,
    &terrainFandB.difRefl[0]);
glUniform4fv(glGetUniformLocation(programId, "terrainFandB.specRefl"), 1,
    &terrainFandB.specRefl[0]);
glUniform4fv(glGetUniformLocation(programId, "terrainFandB.emitCols"), 1,
    &terrainFandB.emitCols[0]);
glUniform1f(glGetUniformLocation(programId, "terrainFandB.shininess"),
    terrainFandB.shininess);

glUniform4fv(glGetUniformLocation(programId, "globAmb"), 1, &globAmb[0]);

glUniform4fv(glGetUniformLocation(programId, "light0.ambCols"), 1,
    &light0.ambCols[0]);
glUniform4fv(glGetUniformLocation(programId, "light0.difCols"), 1,
    &light0.difCols[0]);
glUniform4fv(glGetUniformLocation(programId, "light0.specCols"), 1,
    &light0.specCols[0]);
glUniform4fv(glGetUniformLocation(programId, "light0.coords"), 1,
    &light0.coords[0]);
```

Back in the setup function, the light and terrain materials are applied to the corresponding positions in the vertex and fragment shaders.



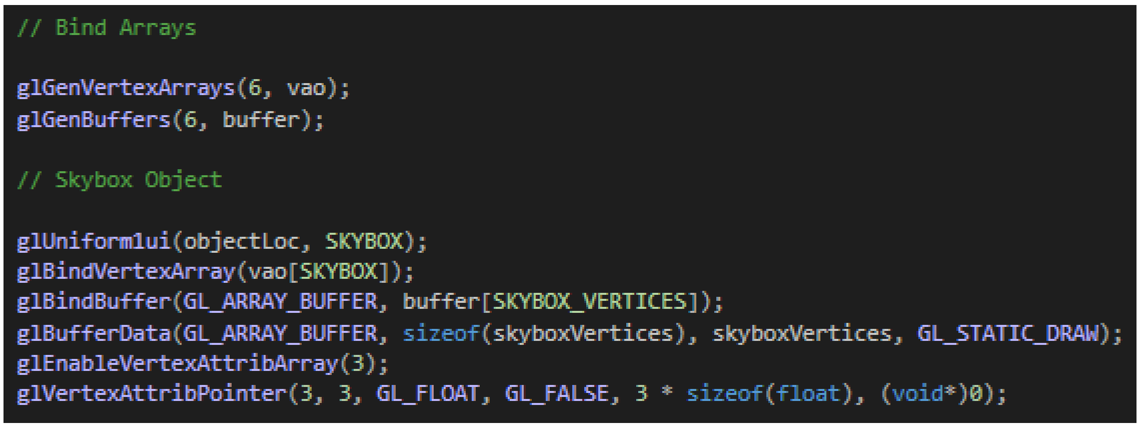
```
// Create textures

image[0] = getbmp("./Textures/grass.bmp"); // Load image
image[1] = getbmp("./Textures/rock.bmp");
image[2] = getbmp("./Textures/sand.bmp");
image[3] = getbmp("./Textures/water.bmp");
image[4] = getbmp("./Textures/sky.bmp");

// Grass

glGenTextures(1, texture); // Texture ID

glActiveTexture(GL_TEXTURE0); // Activate Texture
glBindTexture(GL_TEXTURE_2D, texture[0]); // Bind Grass image
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, image[0]->sizeX, image[0]->sizeY, 0, GL_RGBA, GL_UNSIGNED_BYTE, image[0]->data); // Set up texture map
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glGenerateMipmap(GL_TEXTURE_2D);

grassTexLoc = glGetUniformLocation(programId, "grassTex"); // Send to fragment shader
glUniform1i(grassTexLoc, 0);
```

Next, all of the bitmap files are loaded using the getbmp file, and their values are applied to individual textures. The textures are bound into the texture array, and a texture map is generated from the image data. This is then passed onto the fragment shader. This is the same for all of the materials apart from the skybox cubemap.



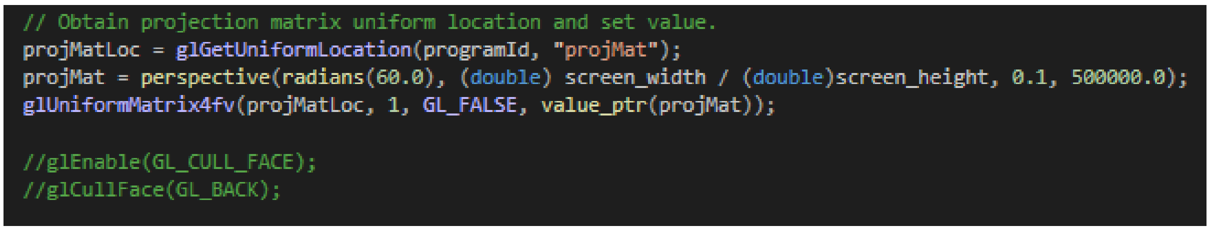
```
std::string skyboxTextures[] =
{
    "Textures/skyright.png",
    "Textures/skyleft.png",
    "Textures/skytop.png",
    "Textures/skybottom.png",
    "Textures/skyfront.png",
    "Textures/skyback.png"
};
glUseProgram(programId);
glGenTextures(4, texture);
glActiveTexture(GL_TEXTURE3);
glBindTexture(GL_TEXTURE_CUBE_MAP, texture[3]);

int width, height;
unsigned char* data1 = SOIL_load_image(skyboxTextures[0].c_str(), &width, &height, 0, SOIL_LOAD_RGBA);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + 0, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data1);

unsigned char* data2 = SOIL_load_image(skyboxTextures[1].c_str(), &width, &height, 0, SOIL_LOAD_RGBA);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + 1, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data2);

unsigned char* data3 = SOIL_load_image(skyboxTextures[2].c_str(), &width, &height, 0, SOIL_LOAD_RGBA);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + 2, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data3);

unsigned char* data4 = SOIL_load_image(skyboxTextures[3].c_str(), &width, &height, 0, SOIL_LOAD_RGBA);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + 3, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data4);

unsigned char* data5 = SOIL_load_image(skyboxTextures[4].c_str(), &width, &height, 0, SOIL_LOAD_RGBA);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + 4, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data5);

unsigned char* data6 = SOIL_load_image(skyboxTextures[5].c_str(), &width, &height, 0, SOIL_LOAD_RGBA);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + 5, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data6);

glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
glGenerateMipmap(GL_TEXTURE_2D);

skyboxTexLoc = glGetUniformLocation(programId, "skyboxTexture");
glUniform1i(skyboxTexLoc, 3);
```

Here, there are image files for each side of the skybox. These are loaded through an external lib called SOIL, and their data is added to each side of a cube map texture. Then, like before, mip maps are generated to prevent flickering at distance, the wrapping is set and the cubemap is passed onto the fragment shader.



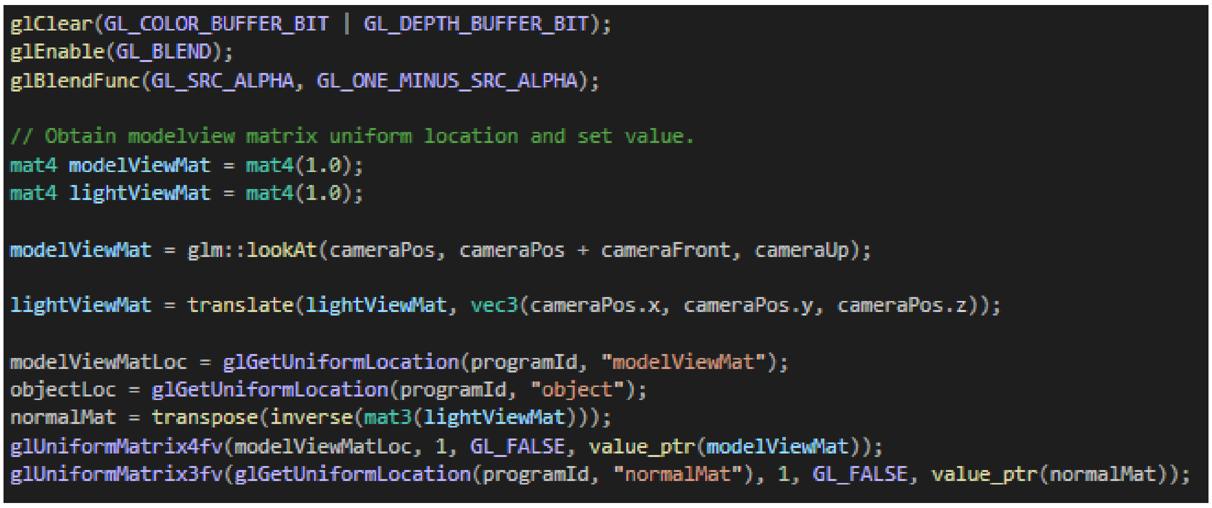
```
// Bind Arrays

glGenVertexArrays(6, vao);
glGenBuffers(6, buffer);

// Skybox Object

glUniform1ui(objectLoc, SKYBOX);
glBindVertexArray(vao[SKYBOX]);
glBindBuffer(GL_ARRAY_BUFFER, buffer[SKYBOX_VERTICES]);
glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices), skyboxVertices, GL_STATIC_DRAW);
glEnableVertexAttribArray(3);
glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
```

Next, the objects are bounded, just like in the 'CreateTerrainObject' function earlier. This is why the terrain is not one of the object here, as it has already been bound in the function. Each vertex is added to the buffer, and then sent to the vertex shader.



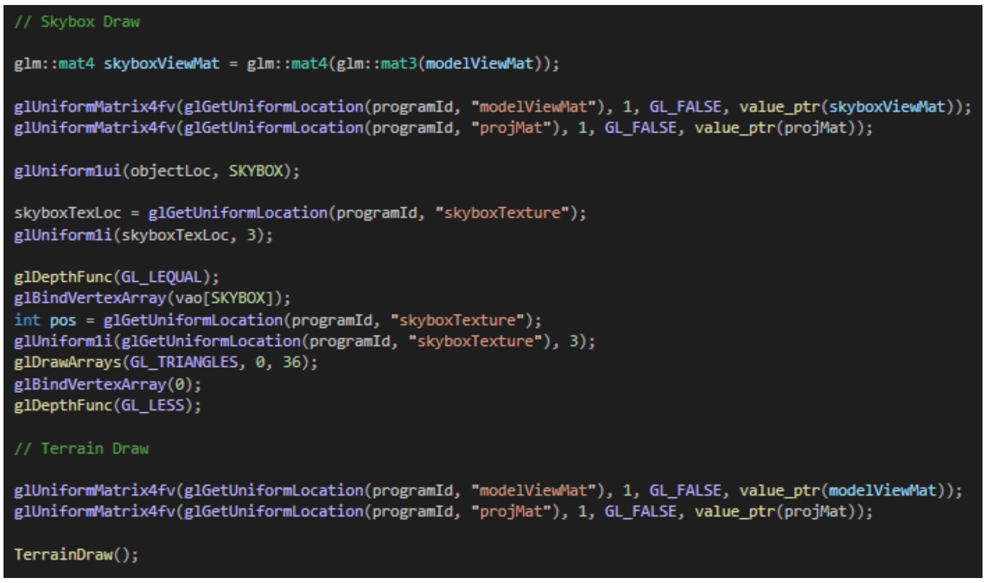
```
// Obtain projection matrix uniform location and set value.
projMatLoc = glGetUniformLocation(programId, "projMat");
projMat = perspective(radians(60.0), (double) screen_width / (double)screen_height, 0.1, 500000.0);
glUniformMatrix4fv(projMatLoc, 1, GL_FALSE, value_ptr(projMat));

//glEnable(GL_CULL_FACE);
//glCullFace(GL_BACK);
```

As the last part of the setup function, I obtain a projection matrix and pass it onto the vertex shader to be used for object positioning. I also am not using back face culling, as whilst it should decrease the amount of processing by double by removing the bottom of every object, it does create strange artefacts in the clouds and water due to their transparency. This may be solvable by not using a triangle strip, but rather a rectangle however I took the performance decrease in order to make the objects render correctly.



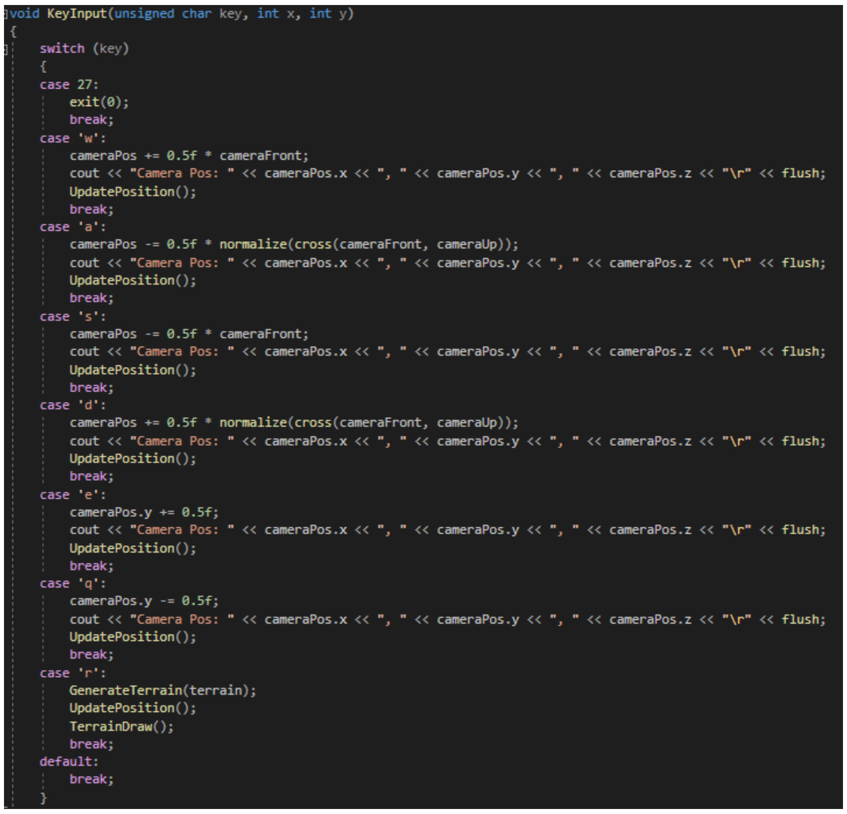
```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

// Obtain modelview matrix uniform location and set value.
mat4 modelViewMat = mat4(1.0);
mat4 lightViewMat = mat4(1.0);

modelViewMat = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

lightViewMat = translate(lightViewMat, vec3(cameraPos.x, cameraPos.y, cameraPos.z));

modelViewMatLoc = glGetUniformLocation(programId, "modelViewMat");
objectLoc = glGetUniformLocation(programId, "object");
normalMat = transpose(inverse(mat3(lightViewMat)));
glUniformMatrix4fv(modelViewMatLoc, 1, GL_FALSE, value_ptr(modelViewMat));
glUniformMatrix3fv(glGetUniformLocation(programId, "normalMat"), 1, GL_FALSE, value_ptr(normalMat));
```

Next is the render function. This function is refreshed by Glut, and is in charge of all of the object positions and drawing. First, a model view matrix and light view matrix is initialised. The view matrix is where objects are projected onto the monitor from, therefore it used a 'lookAt' function from the camera position towards the camera's front direction vector in respect to an up vector. Meanwhile, the light view matrix is simply translated by the camera's position. This is because turning the camera should not affect how the light is displayed on an object, so the light matrix is used for the normals whereas the model matrix is used for object positions. These are sent to the shaders.



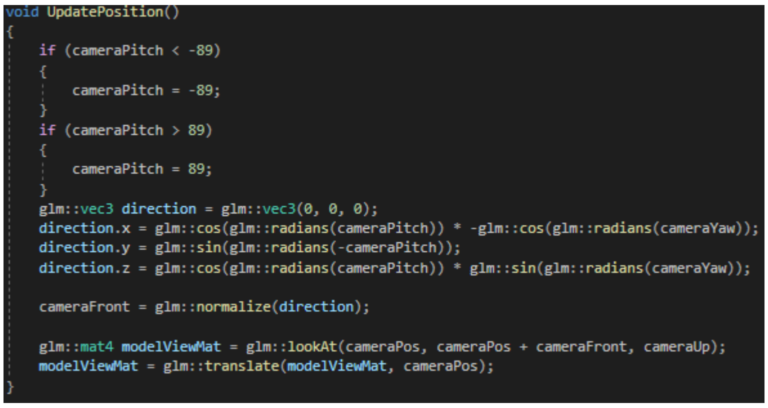
```
// Skybox Draw

glm::mat4 skyboxViewMat = glm::mat4(glm::mat3(modelViewMat));

glUniformMatrix4fv(glGetUniformLocation(programId, "modelViewMat"), 1, GL_FALSE, value_ptr(skyboxViewMat));
glUniformMatrix4fv(glGetUniformLocation(programId, "projMat"), 1, GL_FALSE, value_ptr(projMat));

glUniform1ui(objectLoc, SKYBOX);

skyboxTexLoc = glGetUniformLocation(programId, "skyboxTexture");
glUniform1i(skyboxTexLoc, 3);

glDepthFunc(GL_LEQUAL);
glBindVertexArray(vao[SKYBOX]);
int pos = glGetUniformLocation(programId, "skyboxTexture");
glUniform1i(glGetUniformLocation(programId, "skyboxTexture"), 3);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
glDepthFunc(GL_LESS);

// Terrain Draw

glUniformMatrix4fv(glGetUniformLocation(programId, "modelViewMat"), 1, GL_FALSE, value_ptr(modelViewMat));
glUniformMatrix4fv(glGetUniformLocation(programId, "projMat"), 1, GL_FALSE, value_ptr(projMat));

TerrainDraw();
```

The skybox uses a different draw method than the rest of the objects, as it is a cube map. First, the object is designated as the skybox. It's texture is passed onto the fragment shader, and then the depth function is set to 'GL_LEQUAL'. This makes it so that the skybox is drawn behind every object, as when the incoming depth value is less than what is stored, the depth function passes. The object is drawn using 'glDrawArrays', Additionally, the skybox does not use the view model matrix, but its own skybox view matrix, which is turned from a 4 by 4 matrix to a 3 by 3 matrix and then back again. This removes its translation values, making sure that the camera never gets closer to the skybox to give it an illusion of scale. The 'TerrainDraw' function is simply the same as the rest but placed into a function so that it can be called separately whenever a new terrain is generated. With this function complete, the program draws a skybox behind a terrain with animated water and clouds, as well as multiple materials for the terrain. The last functions are those in charge of keyboard and mouse movement.



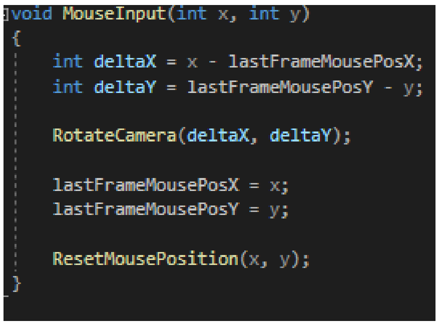
```cpp
void KeyInput(unsigned char key, int x, int y)
{
    switch (key)
    {
    case 27:
        exit(0);
        break;
    case 'w':
        cameraPos += 0.5f * cameraFront;
        cout << "Camera Pos: " << cameraPos.x << ", " << cameraPos.y << ", " << cameraPos.z << "\r" << flush;
        UpdatePosition();
        break;
    case 'a':
        cameraPos -= 0.5f * normalize(cross(cameraFront, cameraUp));
        cout << "Camera Pos: " << cameraPos.x << ", " << cameraPos.y << ", " << cameraPos.z << "\r" << flush;
        UpdatePosition();
        break;
    case 's':
        cameraPos -= 0.5f * cameraFront;
        cout << "Camera Pos: " << cameraPos.x << ", " << cameraPos.y << ", " << cameraPos.z << "\r" << flush;
        UpdatePosition();
        break;
    case 'd':
        cameraPos += 0.5f * normalize(cross(cameraFront, cameraUp));
        cout << "Camera Pos: " << cameraPos.x << ", " << cameraPos.y << ", " << cameraPos.z << "\r" << flush;
        UpdatePosition();
        break;
    case 'e':
        cameraPos.y += 0.5f;
        cout << "Camera Pos: " << cameraPos.x << ", " << cameraPos.y << ", " << cameraPos.z << "\r" << flush;
        UpdatePosition();
        break;
    case 'q':
        cameraPos.y -= 0.5f;
        cout << "Camera Pos: " << cameraPos.x << ", " << cameraPos.y << ", " << cameraPos.z << "\r" << flush;
        UpdatePosition();
        break;
    case 'r':
        GenerateTerrain(terrain);
        UpdatePosition();
        TerrainDraw();
        break;
    default:
        break;
    }
```

This function simply takes the users input and updates the camera position depending on the key pressed. Up and down adds straight to the y value so that no matter the facing, the camera always goes up and down, whereas the other move functions use the camera orientation.



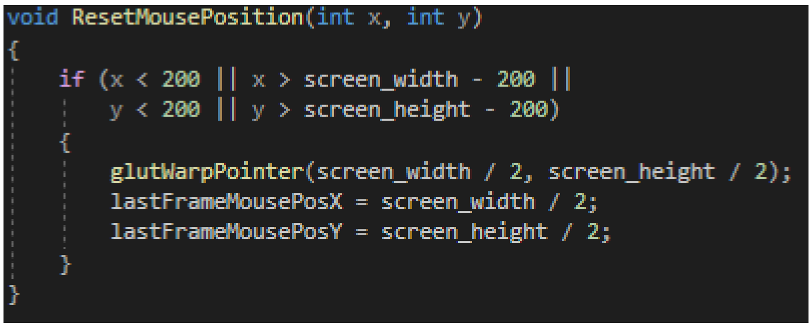
```cpp
void UpdatePosition()
{
    if (cameraPitch < -89)
    {
        cameraPitch = -89;
    }
    if (cameraPitch > 89)
    {
        cameraPitch = 89;
    }
    glm::vec3 direction = glm::vec3(0, 0, 0);
    direction.x = glm::cos(glm::radians(cameraPitch)) * -glm::cos(glm::radians(cameraYaw));
    direction.y = glm::sin(glm::radians(-cameraPitch));
    direction.z = glm::cos(glm::radians(cameraPitch)) * glm::sin(glm::radians(cameraYaw));

    cameraFront = glm::normalize(direction);

    glm::mat4 modelViewMat = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
    modelViewMat = glm::translate(modelViewMat, cameraPos);
}
```

The update position function clamps the camera from looking too far up or down, and sets the 'cameraFront' direction based on the camera pitch and yaw values. It also updated the model view matrix with this new position.



```
void MouseInput(int x, int y)
{
    int deltaX = x - lastFrameMousePosX;
    int deltaY = lastFrameMousePosY - y;

    RotateCamera(deltaX, deltaY);

    lastFrameMousePosX = x;
    lastFrameMousePosY = y;

    ResetMousePosition(x, y);
}
```

The mouse input function takes the x and y position of the mouse cursor, comparing them to the position in the previous frame and applying a rotation to the camera based on that value. Therefore if there is no difference, that means the cursor is standing still, thus the camera is not updated. It also resets the mouse position to the middle so that the view can be moved around without falling off the window. It does this by clamping x and y to a value taken away from the screen width and height, and setting the cursor position to the middle of the screen.



```
void ResetMousePosition(int x, int y)
{
    if (x < 200 || x > screen_width - 200 ||
        y < 200 || y > screen_height - 200)
    {
        glutWarpPointer(screen_width / 2, screen_height / 2);
        lastFrameMousePosX = screen_width / 2;
        lastFrameMousePosY = screen_height / 2;
    }
}
```

Evaluation

Overall, I am quite pleased with my final result, as I have a smooth terrain generated using Perlin noise and various other elements in the scene such as animated water, multiple textures, and 3D camera controls. It was definitely challenging to grasp the idea of the Square-Algorithm, and rendering several objects in the scene, however after that it was a lot of experimenting to make it look like and behave how I wanted. I would have liked to do a lot more, such as generating trees and grass, more lighting such as reflections on the water and possibly even generating more terrain when it runs out.

Furthermore, a simple improvement would have been to separate everything into functions and classes, which would allow me to add as many objects as I wanted without re-adding the same code repeatedly. This would make it a lot less cluttered and speed up my development time.

Another thing I would like to have tried is generating more detailed clouds using Perlin noise. Rather than just changing the height map, I would like to have tried changing the transparency of all of the vertices to some areas clear and some areas more cloudy.

I found this module very interesting and would like to add these additional elements in the future to use for my portfolio.

References

External libraries used:

GLM

SOIL

Getbmp

Libnoise (noise, noiseutils)

freeGlut

Textures:

Grass texture - http://texturelib.com/texture/?path=/Textures/grass/grass/grass_grass_0131

Stone texture - http://texturelib.com/texture/?path=/Textures/rock/stones/rock_stones_0185

Sand texture - http://texturelib.com/texture/?path=/Textures/soil/sand/soil_sand_0040

Seawater texture - http://texturelib.com/texture/?path=/Textures/water/water/water_water_0051

Sky texture - https://3djungle.net/textures/sky/5926/

Skybox texture - https://www.cleanpng.com/png-skybox-texture-mapping-cube-mapping-desktop-wallpa-6020000/