5CM507 – Part2, Week 9 Tutorial

Creating a Sky Box Node, adding fog and detecting the height of terrain

Introduction

Now you have terrain, it would be better if the area above the terrain was more realistic. This tutorial talks you through what you need to do create a node that can display a sky box. In addition, you will add a method to your terrain node that returns the height of the terrain at a particular point. This is in preparation for next week when we will look at detecting collisions between objects.

In the files this week, you nave been provided a shader that can be used to render the sky box (SkyShader.hsls), as well as a texture cube map that represents the sky (skymap.dds).

Many of the steps are very similar to the nodes you have created already and are not repeated. All that will be discussed here is the areas specific to displaying a sky map. How this works is covered in the lecture this week.

1. Create a New Node called SkyNode

The constructor for this node should take as parameters the name of the node, the name of the sky box texture file and the radius of the sphere that the sky box is rendered on (I used a radius of 5000).

All of the steps for the initialisation of the node are the same as you have done previously and so will not be repeated here.

The vertex structure you need to use for this node only needs to hold the position of each vertex, i.e.

```
struct Vertex
{
    Vector3 Position;
};
```

The constant buffer only needs to pass in the combined transformation matrix, i.e.

```
struct CBUFFER
{
    XMMATRIX    CompleteTransformation;
};
```

The geometry you need for the node, i.e. the vertices and indices, is that of a sphere surrounding the scene. A method to create a sphere is included in the file skycode.cpp. For the tessellation parameter, I used a value of 30.

There are two other things you need to setup. You need to setup renderer states to turn off back face culling and re-enable it. You also need code to change the depth buffer settings to be less-than-equal-to instead of less-than, as well as code to reset it back to the defaults. Code to do these is also provided in skycode.cpp.

Use CreateDDSTextureFromFile to load the texture cube into memory.

2. Rendering the Sky Cube

The steps for rendering the sky cube are as follows:

- Calculate the world transformation for the sphere that the sky map is rendered on to. This should simply be a translation based on the position of the camera. This will ensure that the sphere is always centered at the camera position, so you never reach the edge of the sphere.
- Now you can calculate the complete world-view-projection transformation using this world transformation. We calculate the complete transformation as we have done before, but we now need to calculate the transpose of that matrix.
- Setup the constant buffer with the transposed matrix.
- You pass the sky map texture through to the shader using PSSetShaderResources just like you have done previously.
- Before the call to DrawIndexed, you need to disable backface culling and change the depth stencil equation. This is done using the following two calls:

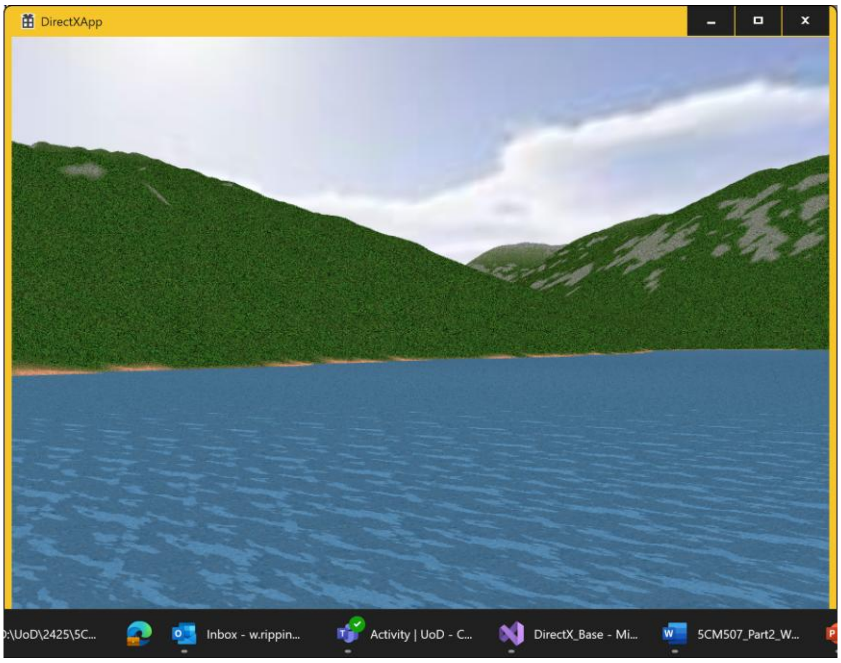
```
_deviceContext->RSSetState(_noCullRasteriserState.Get());
_deviceContext->OMSetDepthStencilState(_stencilState.Get(), 1);
```

- After the call to DrawIndexed, re-enable back face culling and set the depth stencil values back to the defaults using:

```
_deviceContext->OMSetDepthStencilState(nullptr, 1);
_deviceContext->RSSetState(_defaultRasteriserState.Get());
```

3. Add the SkyNode to your scene graph DirectXApp.cpp.

Once you have created your new node, you now need to create a new instance of the node and add it to the scene graph in DirectXApp.cpp. When you run your code now, you will hopefully see a realistic sky representation above your terrain.



3. Adding Fog

If you wish to add fog, to your scene, you can do it using the calculations shown on the slides in the lecture. You can either hard-code the values for the fog colour, fog start and fog range into your shaders or, to be more flexible, extend the constant buffer and pass the values in to the shader using the constant buffer.

4. Getting the Height of Terrain

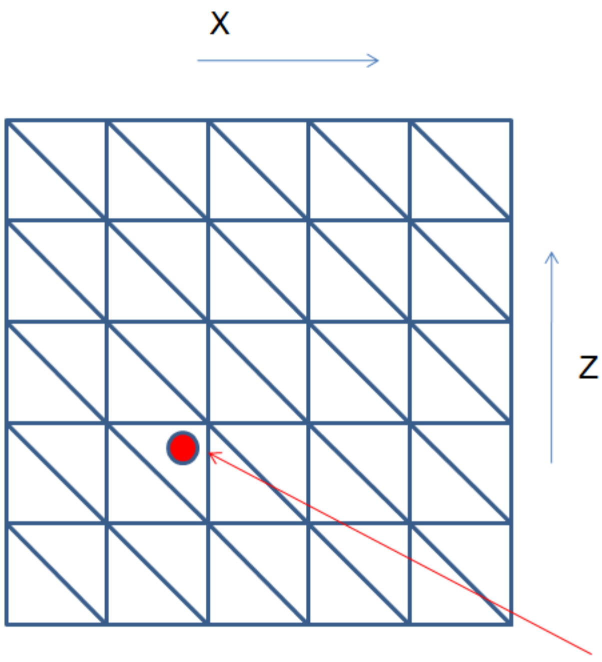
In the past few weeks, you have seen how to generate terrain and how to move a camera around a scene. We now need to look at how we can determine if we have hit something. We will start with the terrain. As you will have seen, at the moment, you can fly underneath the terrain which is usually not a desirable thing to do.

In order to prevent this, we need a way of determining the height (i.e. the y value) for any position in the world (i.e. for any x and z value). You need to add a method to your TerrainNode that returns the y value for any given x and z value. It might have a signature something like the following:

```
float GetHeightAtPoint(float x, float z);
```

The following steps will take you through what you need to do in this method

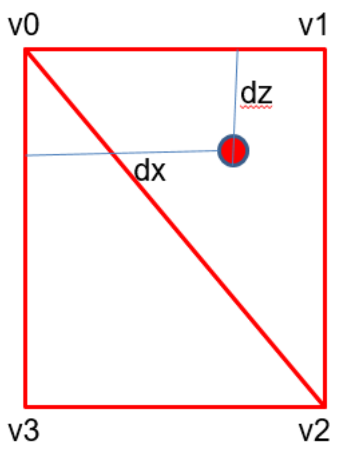
Remember that our terrain is made up of squares that have the same width and each square is divided up into two triangles. Imagine that the red dot in the grid below is the position we are interested in.



The first thing we need to calculate is which cell in the grid contains the x, z value we are interested in. If we maintain the start x and z values of the terrain and the width/height of a cell, then:

```
int cellX = static_cast<int>((x - _terrainStartX) / _spacing);
int cellZ = static_cast<int>((_terrainStartZ - z) / _spacing);
```

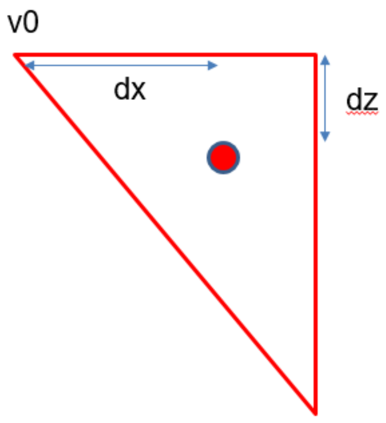
Once we have identified the cell the point x, z is in, we now need to identify which of the triangles we are interested in.



If dz is the difference in the z value between the point and the edge of the grid cell and dx is the difference in the x value between the point and the edge of the grid cell, then:

```
If dx > dz
    We are in triangle with vertices v0, v1, v2
Else
    We are in triangle with vertices v0, v2, v3
```

Once we have identified which triangle contains the point x, z, we now need to identify the y value at that point. To do this, we make use of the fact that the dot product of a vector on a plane and the planes' normal vector gives a value of 0 (you saw earlier in the module).



$$N.(P - v0) = 0$$

Where:	N is the normal vector for the triangle
	P is the point we are interested in
	v0 is one of the vertices of the triangle

Expanding out the dot product equation, we get:

$$N_x(P_x - v0_x) + N_y(P_y - v0_y) + N_z(P_z - v0_z) = 0$$

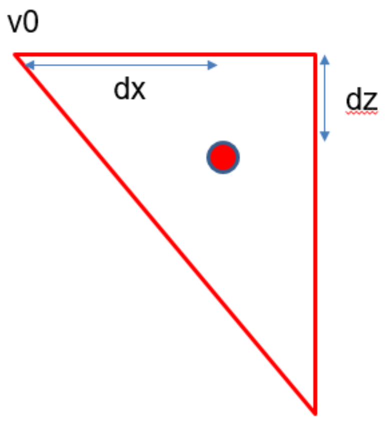
We extract the y components by moving them to the other side and dividing by Ny:

$$P_y - v0_y = (N_x(P_x - v0_x) + N_z(P_z - v0_z)) / -N_y$$

We can now rearrange so that we have Py on its own:

$$P_y = (N_x(P_x - v0_x) + N_z(P_z - v0_z)) / -N_y + v0_y$$

So, for the given dx and dz inside the triangle:



$$P_y = v0_y + \frac{(N_x * dx + N_z * dz)}{-N_y}$$

Testing Your Method

One way of testing your method is by flying your camera over the terrain from one x, z value to another, adjusting the x and z values for the camera in each call to DirectXApp::UpdateSceneGraph(). You can then work out the y value at that point and set the position of the camera based on the x, y and z coordinates (you might want to add a small adjustment to y so that you are not skimming the surface). You should not fly into the terrain as you fly between the two points.