Graphics – Work for Part 2, Week 6

The work this week requires quite a bit of careful thinking. There is nothing new in the DirectX calls you will be making, but you will need to implement quite a few algorithms to generate the terrain so make sure you read through this tutorial document carefully.

The goal this week is to start to create a TerrainNode class that inherits from SceneNode. The steps you will do this week are:

- Create a grid that will form the terrain
- Apply a height map to the terrain
- Calculate the normal for the terrain

We will ultimately be applying blended textures to the terrain to simulate different materials, but we will not get there this week.

If you want to take this further in the final assignment and try using techniques such as real-time procedural generation rather than using a height map, I encourage you to try this. However, the core ideas you see here will apply.

An example height map has been provided for you, but there is nothing to stop you creating your own.

Step 1 – Create a TerrainNode class that inherits from SceneNode.

The two core parameters that you will be passing in to your TextureNode constructor will be the name of the node and the name of the file that contains the height map. You might choose to add other parameters such as the size of the grid and the maximum height value, but that is up to you.

By the end of this week, the Initialise method for TerrainNode will need to perform the following steps:

1. Load the height map
2. Generate the vertices and indices for the polygons in the terrain grid
3. Generate the normals for the polygons
4. Create the vertex and index buffers for the terrain polygons.

You will do these in the following steps.

At this point, I would suggest you adjust the eye position used in creating the view matrix so that it is further back and higher. This is set in the constructor for Camera in camera.cpp. Possible values would be:

```
_eyePosition = Vector3(0.0f, 100.0f, -500.0f);
```

Step 2 – Generate the Vertices and Indices

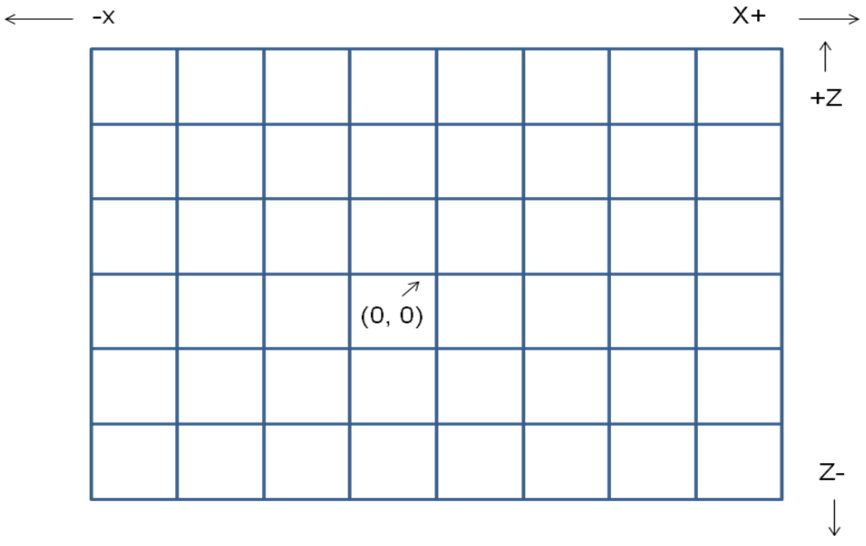
For this week, I would suggest using a copy of the same Vertex structure as used last week in ResourceManager.h which contains a position, a normal and a texture coordinate for each vertex. However, you should give it a different name in your terrain node since we will be adding to it later.

We need to create a mesh that represents a grid that is made up of adjacent square cells. Because we are ultimately going to texture each cell in the grid individually, the four vertices that make up each square cannot be used in other squares (because the texture coordinates would be different). Each square will be a fixed number of pixels on each side.

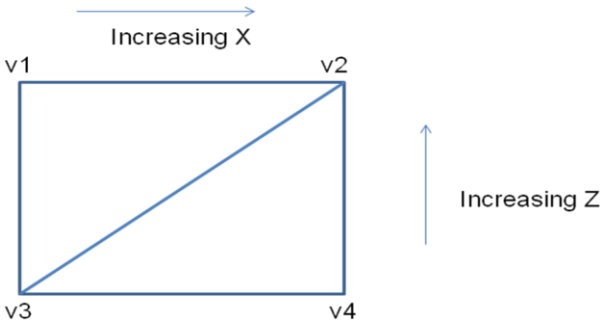
The height map we will use is 1024 pixels by 1024 pixels. Each pixel represents the height at a vertex in the grid. Therefore, the grid will consist of 1023 rows by 1023 columns. The grid will be built such that x = 0 and z = 0 is in approximately the centre of the grid. If you use a width for each cell of 10 pixels, the lowest x value will be -5120 and the largest will be 5110. The z value will also vary from a large positive z value at the start of the grid (furthest into the screen) decreasing to a large negative z value.

To build the grid, use a pair of nested loops, adding vertices to a vector as you go. At this stage, you should set the normal for each vector to (0,0,0) – you will calculate the normals later. The X and Z coordinates can be calculated from the loop values. For now, set Y to 0. You will change this later once we load in the height map. Set the Texture coordinates to (0,0), (0, 1), (1, 0) and (1, 1) for the four vertices for each square.

By the end, you should have vertices for the following grid where the vertices are the corners of the squares.



Once you have created the vertices for each square, you need to create the indices. splitting the grid into two triangles per square. To get the correct winding order so that the terrain is facing upwards, you need to divide each square up as follows:



For each square in the grid that has vertices v1, v2, v3 and v4, the indices for triangle 1 point to v1, v2 and v3 and the indices for triangle 2 point to v3, v2 and v4 in that order. Do this for every square in the grid, adding six indices to the vector of indices.

Once you have created the grid, you can try getting it to display. You will need to create the vertex and index buffers (you have seen code that does this in all of the previous examples given to you). If you have used C++ vectors to hold the vertices and indices, then you should use the address of the first element in the vector as the address of data to be used to initialise the buffers. For example, if you set them up in a vector called _vertices, the D3D11_SUBRESOURCE_DATA structure used for the initialisation data, would be as follows:

```
D3D11_SUBRESOURCE_DATA vertexInitialisationData;
vertexInitialisationData.pSysMem = &_vertices[0];
```

Your starting point for the shaders for the terrain should be the same as those you used in week 5, but I would suggest making a copy of them and putting them in a separate file as you will be changing them later.

If you want to display the grid as wireframe, rather than solid shaded (which is very useful for seeing if you did it right), then I have provided the code for a method called BuildRendererStates in the file RenderStates.c. Add this code to your TerrainNode class and call it from your initialise method. Also, add the following definitions to TerrainNode.h:

```
ComPtr<ID3D11RasterizerState>    _defaultRasteriserState;
ComPtr<ID3D11RasterizerState>    _wireframeRasteriserState;
```

Now, in your Render method, before you call DrawIndexed, set the renderer state to wireframe mode using the following call:

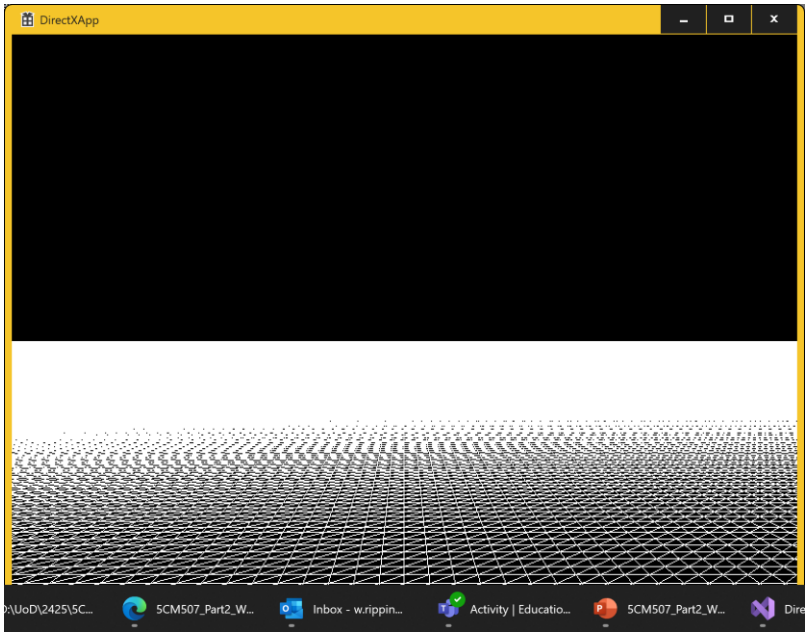
```
_deviceContext->RSSetState(_wireframeRasteriserState.Get());
```

While you are using wireframe rendering, I would recommend just changing the pixel colour to be white, i.e. change the pixel shader to simply be:

```
float4 PShader(PixelShaderInput input) : SV_TARGET
{
	float4 colour = float4(1.0f, 1.0f, 1.0f, 1.0f);
```

```
        return colour;
}
```

Once you have written all of the code for TerrainNode, add a new TerrainNode to the scene graph in CreateSceneGraph. When you run your code now, you should hopefully see a flat grid stretching into the distance that looks something like the following:



Step 3 – Loading and Using the Height Map

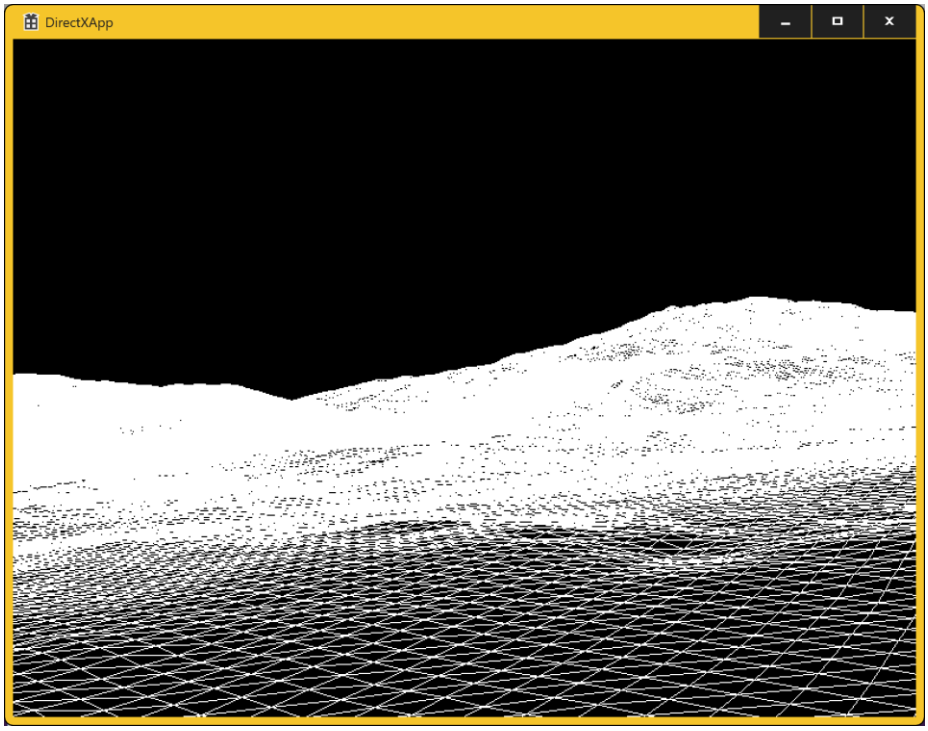
Code to read the height map has been provided in LoadHeightMap.c. You can use this or provide your own. This code reads the values from the height map and normalises them into floating point values between 0 and 1. These are stored in a vector of floats defined as follows:

```
        vector<float>        _heightValues;
```

The code I have provided also assumes there are two unsigned int variables declared in your class called _numberOfXPoints and _numberOfZPoints that are set to the number of vertices in each direction (in this case, 1024).

Now you should update your code that creates the grid so that it takes the Y values for each vertex from the height map. Because these are now normalised values, you should multiply the value in the height map by a suitable maximum world height value (I would suggest 1024 as a starting point) to give a suitable Y value. Note that the height map gives the height value for where the squares meet in the grid, so you need to be careful when determining which Y value to use for each vertex. For most vertices in the grid, except for the ones on the edges of the grid, you will use each height value four times.

Once you have done this, you can rerun your code and you should see a wireframe rendering of hills that looks something like the following:

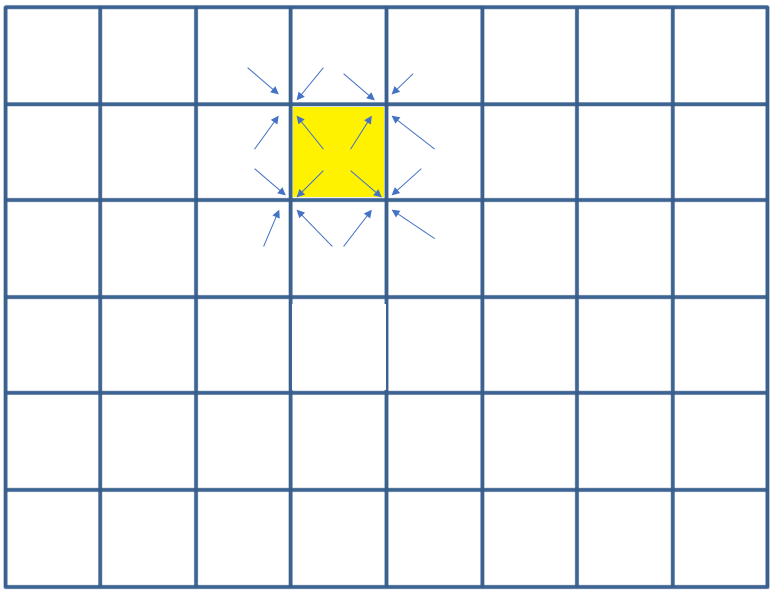


Step 4 – Calculating the Normals

Now you have the grid with Y values, you can calculate the normals for the vertices and update the array of vertices with the calculated normals. The way you do this is a variation on how you created vertex normals earlier in this part of the module. Remember that you set the vertex normal to (0,0,0) when you created the basic grid earlier.

Write a nested loop that iterates through every square in the grid. You will need to calculate the polygon normal for each of the polygons that make up the square. You saw how to create the normal earlier in this part of the module, so I will not repeat it here, but ask me for help if you get stuck. Be careful to ensure that you calculate the two polygon normals so that they are roughly in the same direction. If you end up calculating them so that one points roughly upwards and the other one points roughly downwards, you will end up with a very jagged terrain.

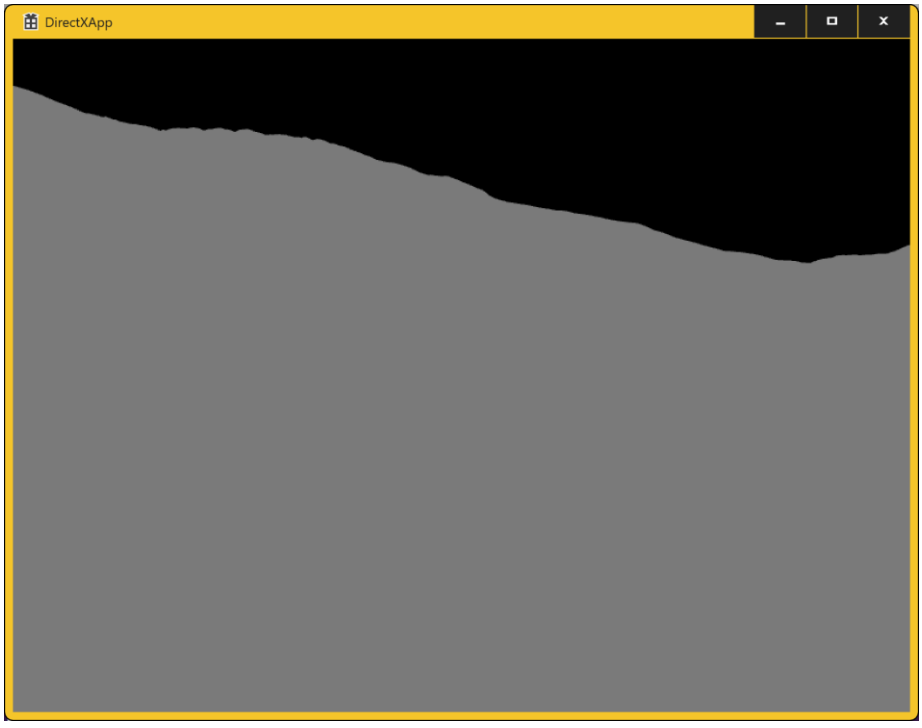
The next step is to add the polygon normals to the vertex normal for each of the vertices that make up the corners of the polygon. However, we are not done yet. We also need to add this polygon normal to each of the vertices of adjacent polygons that touch this one. In the diagram below, the arrows point to all of the vertices that the polygon normals for the yellow square may need to be added to.



Once you have done this for every square in the grid, you should then go iterate through all of the vertices again and normalise the normal vectors. Note that, unlike earlier in this module, we do not divide by the contributing counts since all the polygons are the same size and so will have the same impact on the vertex normal. Simply normalising the normal vectors performs the same task here.

Now, restore the pixel shader to the original value, but make sure that line that calculates the final colour does not sample a texture (since we are not providing one yet).

Now you can remove the code that displays in wireframe mode and display more solid hills (although they will be white).



If you want to try applying a texture, I have provided two example textures that you can use. (dirt.jpg and grass.jpg). However, this still gives a very uniform looking terrain. Obviously, we still need to apply more textures, but before we do that, it would be good if we could move over the terrain. So, next week we will look at adding a moveable camera to our framework. Then we will come back and look at texturing our terrain in more detail.