Graphics Part II: Code Tutorial for Week 6

Dr P. Perakis

Preparation

We have provided a fully working code for the loading of a model and a MeshNode class to render a Plane with texture in the `Exercise_05_2.zip` file. This code can be used as a starting point for the following work, or you can use any working code you have up to now. We have also provided the `Exercise_06_0.zip` file which you can use as starting point. You will also need to use the files provided in the `Exercise_06_Files.zip` file.

Introduction

The work this week requires to implement quite a few algorithms to generate a terrain. The goal is to start to create a `TerrainNode` class that inherits from `SceneNode`.

The steps you will do for this are:

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

We will ultimately be applying blended textures to the terrain to simulate different materials, but we will not get there this week. For now, we are going to use different simple images as terrain textures.

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

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.

The files provided in `Exercise_06_Files.zip` file are as follows:

TerrainNode.h and .cpp	The classes that represent the terrain node, for using it in scene graph.
Example_HeightMap.raw	Height map file for the terrain
white.png	A white texture file for use as a default texture
dirt.jpg; grass.jpg; Wood.png; Concrete.png; RGB_Colormap.png;	Various textures for the terrain

Step 1 – Creating the TerrainNode class that inherits from SceneNode.

The two core parameters that you will be passing into your `TerrainNode` 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.

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 triangles in the terrain grid
3. Generate the normals for the triangles
4. Create the vertex and index buffers for the terrain triangles.

```
bool TerrainNode::Initialise()
{
	// access the device
	_device = DirectXFramework::GetDXFramework()->GetDevice();
	// access the divice context
	_deviceContext = DirectXFramework::GetDXFramework()->GetDeviceContext();

	_useHeightMap = LoadHeightMap(L"Example_HeightMap.raw");
	BuildRendererStates();
	BuildGeometry();
	BuildShaders();
	BuildVertexLayout();
	BuildConstantBuffer();
	BuildTexture(L"white.png");
	//BuildTexture(L"Wood.png");
	//BuildTexture(L"Concrete.png");
	//BuildTexture(L"RGB_Colormap.png");

	return true;
}
```

We are going to do these in the following steps.

Step 2 – Generating the Vertices and Indices

Generation of Vertices and Indices is done in `TerrainNode::BuildGeometry()` function. For the terrain we can use the same `Vertex` and `CBUFFER` structure as used in `ResourceManager.h` which contains a position, a normal and a texture coordinate for each vertex.

In `TerrainNode` class:

```
struct Vertex
{
	Vector3 Position;
	Vector3 Normal;
	Vector2 TexCoords;
};

struct CBUFFER
{
	Matrix		WorldViewProjection;
	Matrix		World;
	Vector4		MaterialColour;
	Vector4		AmbientLightColour;
	Vector4		DirectionalLightColour;
	Vector4		DirectionalLightVector;
};
```

We need to create a mesh that represents a grid that is made up of adjacent square cells. Thus, we need to define the four vertices that make up each square. Each square will then be mapped to a fixed number of pixels on each side according to the texture it is mapped on.

- We start with a grid of squares, each split into two triangles.
- The grid needs to be large enough to cover the area of the scene we are interested in

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.

Here I have used static array structures as follows.

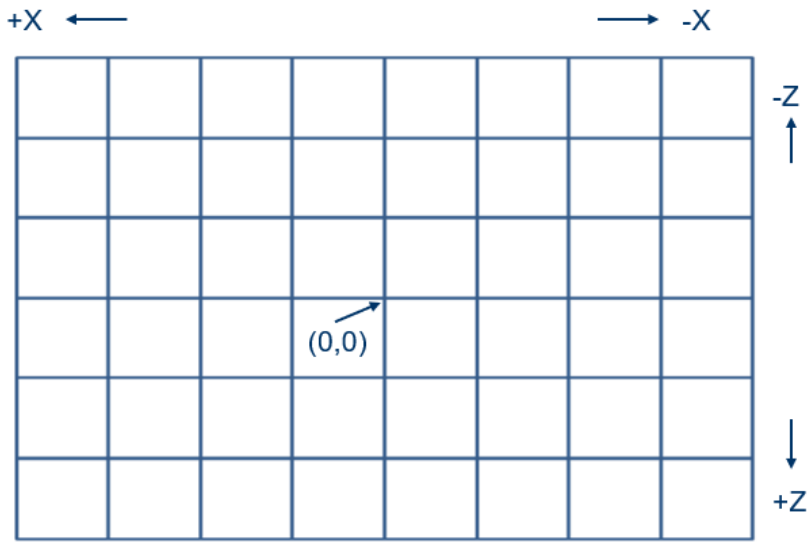
In `TerrainNode` class:

```
const UINT _xnumVert = 257;
const UINT _znumVert = 257;
const UINT _numVert = _xnumVert * _znumVert;
const UINT _numInds = (_xnumVert - 1) * (_znumVert - 1) * 2 * 3;
```

In `TerrainNode::BuildGeometry()` function:

```
struct TerVertex
{
	XMFLOAT3 Pos;
	XMFLOAT3 Norm;
	XMFLOAT2 Tex;
};

TerVertex* terrainVerts = new TerVertex[_numVert];
UINT* terrainInds = new UINT[_numInds];
```

- The grid is generated by code.
- Each square consists of four separate vertices.
- By the end, you should have vertices for the following grid where the vertices are the corners of the squares.



To build the grid, we use a pair of nested loops, adding vertices to the vertices structure as we go. The X and Z coordinates can be calculated from the loop values. For now, we set Y to 0. You will change this later once we load in the height map.

The Texture coordinates are calculated according to some conditions for each vertex.

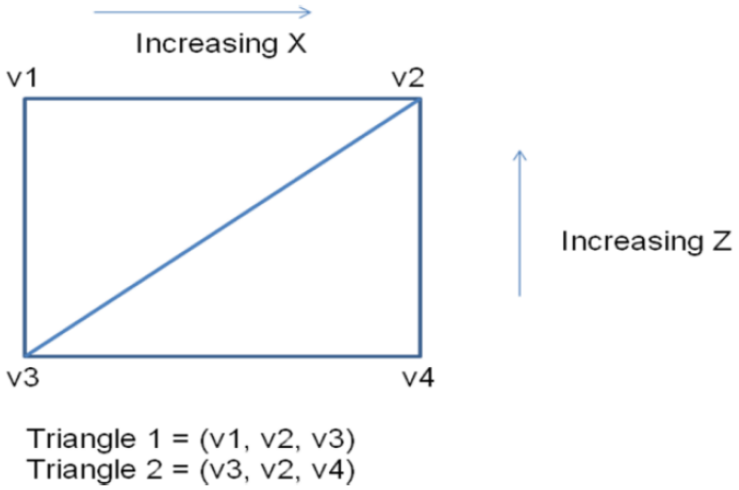
At this stage, you can set the normal for each vertex to (0,1,0) – we will calculate the normals later.

I have parameterized the creation of the grid for being more flexible and be applicable to different situations.

```
float x0 = 0.0f;
float y0 = -80.0f;
float z0 = 0.0f;
float dx = 1.0f;
float dy = 0.1f;
float dz = 1.0f;
float height = 80.0f;

x0 = -((_xnumVert / 2) * dx);
z0 = -((_znumVert / 2) * dz);

//_useHeightMap = false;
srand(0);

for (int j = 0; j < _znumVert; j++)
	for (int i = 0; i < _xnumVert; i++)
	{
		int inx = j * _xnumVert + i;
		terrainVerts[inx].Pos.x = x0 + i * dx;
		if (_useHeightMap)
```

```
                terrainVerts[inx].Pos.y = y0 + _heightValues[inx] * height;
            else
                terrainVerts[inx].Pos.y = y0; // flat surface
            //terrainVerts[inx].Pos.y = terrainVerts[inx].Pos.y + (rand() % 10) *
dy; // add random roughness
            terrainVerts[inx].Pos.z = z0 + j * dz;

            //Normals will be calculated
            terrainVerts[inx].Norm.x = 0.0f;
            terrainVerts[inx].Norm.y = 1.0f;
            terrainVerts[inx].Norm.z = 0.0f;

            //Stretched Texture
            //terrainVerts[inx].Tex.x = i / (float)_xnumVert;
            //terrainVerts[inx].Tex.y = j / (float)_znumVert;

            //Tiling Texture
            terrainVerts[inx].Tex.x = (float)(i % 2);
            terrainVerts[inx].Tex.y = (float)(j % 2);
            //terrainVertices[inx].Tex.x = (float)(i % 8) / 8.0f;
            //terrainVertices[inx].Tex.y = (float)(j % 8) / 8.0f;

            //Random Sampling of Texture
            //terrainVertices[inx].Tex.x = (rand() % 100) / 100.0f;
            //terrainVertices[inx].Tex.y = (rand() % 100) / 100.0f
        }
```

Once we have created the vertices for each square, we 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.

```
for (int j = 0; j < (_znumVert - 1); j++)
        for (int i = 0; i < (_xnumVert - 1); i++)
        {
                int inx = (j * (_xnumVert - 1) + i) * 2 * 3;
```

```
            terrainInds[inx] = j * _xnumVert + i;
            terrainInds[inx + 1] = (j + 1) * _xnumVert + i;
            terrainInds[inx + 2] = j * _xnumVert + i + 1;
            terrainInds[inx + 3] = j * _xnumVert + i + 1;
            terrainInds[inx + 4] = (j + 1) * _xnumVert + i;
            terrainInds[inx + 5] = (j + 1) * _xnumVert + i + 1;
      }
```

Once we 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).

```
// Setup the structure that specifies how big the vertex
// buffer should be
D3D11_BUFFER_DESC vertexBufferDescriptor;
vertexBufferDescriptor.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDescriptor.ByteWidth = sizeof(Vertex) * _numVert;
vertexBufferDescriptor.BindFlags = D3D11_BIND_VERTEX_BUFFER;
vertexBufferDescriptor.CPUAccessFlags = 0;
vertexBufferDescriptor.MiscFlags = 0;
vertexBufferDescriptor.StructureByteStride = 0;

// Now set up a structure that tells DirectX where to get the
// data for the vertices from
D3D11_SUBRESOURCE_DATA vertexInitialisationData;
vertexInitialisationData.pSysMem = terrainVerts;// &terrainVerts;

// and create the vertex buffer
ThrowIfFailed(_device->CreateBuffer(&vertexBufferDescriptor,
&vertexInitialisationData, _vertexBuffer.GetAddressOf()));

// Setup the structure that specifies how big the index
// buffer should be
D3D11_BUFFER_DESC indexBufferDescriptor;
indexBufferDescriptor.Usage = D3D11_USAGE_IMMUTABLE;
indexBufferDescriptor.ByteWidth = sizeof(int) * _numInds;
indexBufferDescriptor.BindFlags = D3D11_BIND_INDEX_BUFFER;
indexBufferDescriptor.CPUAccessFlags = 0;
indexBufferDescriptor.MiscFlags = 0;
indexBufferDescriptor.StructureByteStride = 0;

// Now set up a structure that tells DirectX where to get the
// data for the indices from
D3D11_SUBRESOURCE_DATA indexInitialisationData;
indexInitialisationData.pSysMem = terrainInds;// &terrainInds;

// and create the index buffer
ThrowIfFailed(_device->CreateBuffer(&indexBufferDescriptor,  &indexInitialisationData,
_indexBuffer.GetAddressOf()));
```

Your starting point for the shader for the terrain should be the same as the one you already use.

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 we have provided the code for a method called `BuildRendererStates`.

```
void TerrainNode::BuildRendererStates()
{
      // Set default and wireframe rasteriser states
      D3D11_RASTERIZER_DESC rasteriserDesc;
      rasteriserDesc.FillMode = D3D11_FILL_SOLID;
      //rasteriserDesc.FillMode = D3D11_FILL_WIREFRAME;
```

```
	rasteriserDesc.CullMode = D3D11_CULL_BACK;
	rasteriserDesc.FrontCounterClockwise = false;
	rasteriserDesc.DepthBias = 0;
	rasteriserDesc.SlopeScaledDepthBias = 0.0f;
	rasteriserDesc.DepthBiasClamp = 0.0f;
	rasteriserDesc.DepthClipEnable = true;
	rasteriserDesc.ScissorEnable = false;
	rasteriserDesc.MultisampleEnable = false;
	rasteriserDesc.AntialiasedLineEnable = false;
	ThrowIfFailed(_device->CreateRasterizerState(&rasteriserDesc,
_rasteriserState.GetAddressOf()));
}
```

Call this code in your `initialise` method of the `TerrainNode` class.

Also, add the following definition to `TerrainNode.h`:

```
ComPtr<ID3D11RasterizerState>	_rasteriserState;
```

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

```
// Set the render state
_deviceContext->RSSetState(rasteriserState.Get());
```

While you are using wireframe rendering, I would recommend just changing the material colour to be white:

```
TerrainNode::TerrainNode(wstring name) : SceneNode(name)
{
	_name = name;
	_materialColour = Vector4(1.0f, 1.0f, 1.0f, 1.0f);
	_useHeightMap = false;
}
```

Once you have written all of the code for `TerrainNode`, add a new `TerrainNode` to the scene graph in `CreateSceneGraph`.

Since you are going to use the `TerrainNode` class in `DirectXApp.cpp` just include it, using:

```
#include " TerrainNode.h"
```

And update `CreateSceneGraph()`:

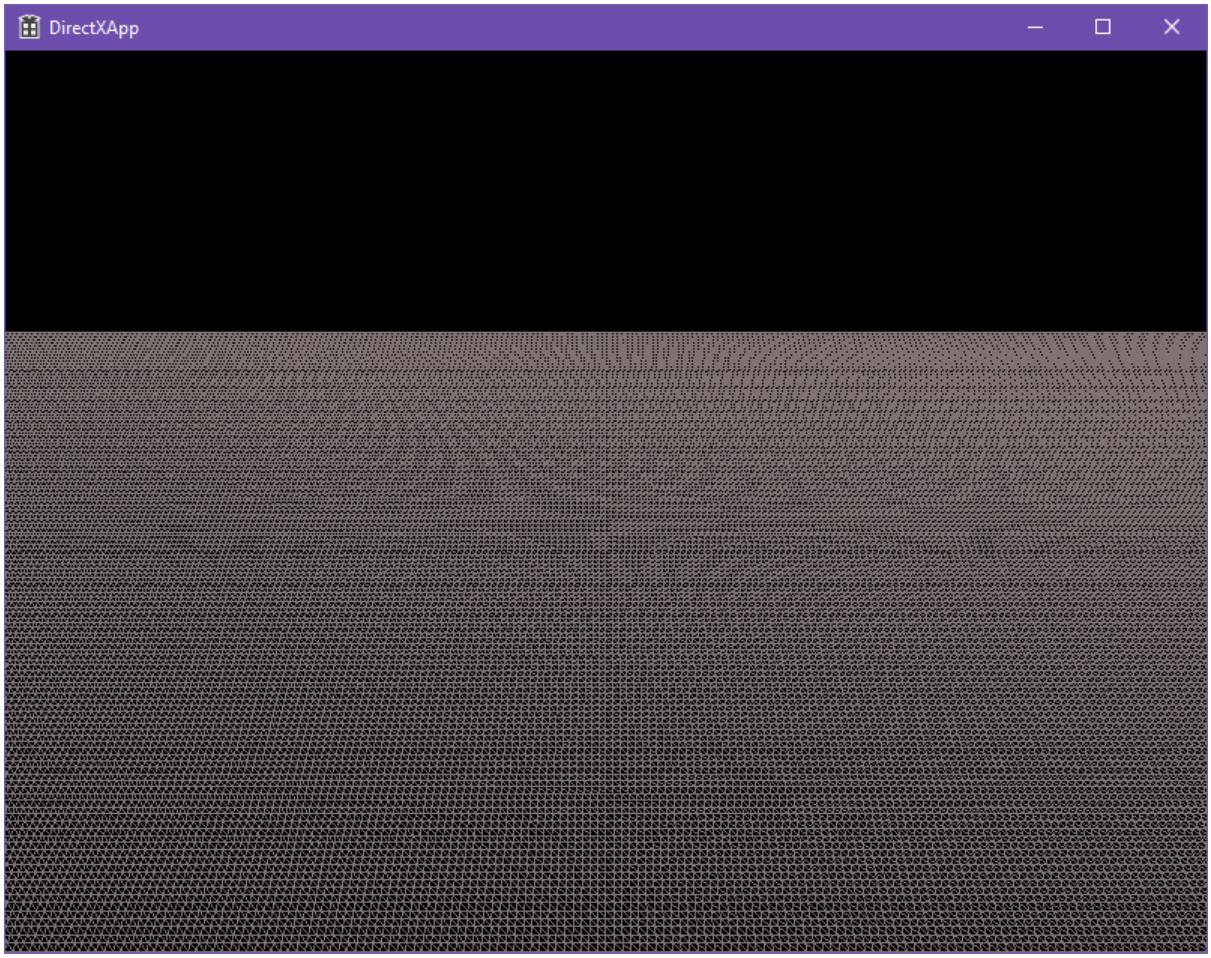
```
void DirectXApp::CreateSceneGraph()
{
	SceneGraphPointer sceneGraph = GetSceneGraph();

	// Add your code here to build up the scene graph

	SceneNodePointer terrainNode_ptr = make_shared<TerrainNode>(L"Terrain");
	terrainNode_ptr->Initialise();
	sceneGraph->Add(terrainNode_ptr);

}
```

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, 50.0f, -50.0f);
//_eyePosition = Vector3(0.0f, 150.0f, -300.0f);
```

The output of the executable

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

The height map we will use is considered an image of maximum size 1024 pixels by 1024 pixels. Each pixel represents the height at a vertex in the grid. Therefore, the grid will ultimately consist of 1023 rows by 1023 columns.

Code to read the height map has been provided in `TerrainNode` class. You can use this or provide your own.

I have provided an enhanced version of this code. Enhanced code reads a subarea of the height map having a size of `_xnumVert * _znumVert`.

The code is the following:

```
bool TerrainNode::LoadHeightMap(const wchar_t* FileName)
{
	//size of Example_HeightMap.raw
	unsigned int numberOfXPoints = 1024;
	unsigned int numberOfZPoints = 1024;
	unsigned int mapSize = numberOfXPoints * numberOfZPoints;  // 1024 * 1024;//

	USHORT* rawFileValues = new USHORT[mapSize];

	ifstream inputHeightMap;
	//inputHeightMap.open(heightMapFilename.c_str(), std::ios_base::binary);
	inputHeightMap.open(FileName, std::ios_base::binary);
	if (!inputHeightMap)
	{
		return false;
	}
	inputHeightMap.read((char*)rawFileValues, mapSize);
	inputHeightMap.close();

	// Normalise BYTE values to the range 0.0f - 1.0f;
	// Saving only what we need from a top-left rectangular area
	for (UINT j = 0; j < _znumVert; j++)
		for (UINT i = 0; i < _xnumVert; i++)
		{
			int inx = j * numberOfXPoints + i;
			_heightValues.push_back((float)rawFileValues[inx] / 65536.0f);
//2 bytes = 256*256 = 65536
		}
	delete[] rawFileValues;
	return true;
}
```

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 in `TerrainNode` class as follows:

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

The code we 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) according to the sizes of the used height map.

Now you should update your code that creates the grid so that it takes the Y values for each vertex from the height map.

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 dx = 10 units, the lowest x value for _xnumVert = 257 will be -1280 units and the largest will be 1280. The z value will also vary from a large positive z value at the start of the grid (furthest out from the screen) decreasing to a large negative z value.

These are parameterized in the following variables:

```
float x0 = 0.0f;
float y0 = -80.0f;
float z0 = 0.0f;
float dx = 1.0f;
float dy = 0.1f;
float dz = 1.0f;
float height = 80.0f;

x0 = -((_xnumVert / 2) * dx);
z0 = -((_znumVert / 2) * dz);
```

The _useHeightMap is a Boolean for using or ignoring the Height map values:

```
if (_useHeightMap)
	terrainVerts[inx].Pos.y = y0 + _heightValues[inx] * height;
else
	terrainVerts[inx].Pos.y = y0; // flat surface
```

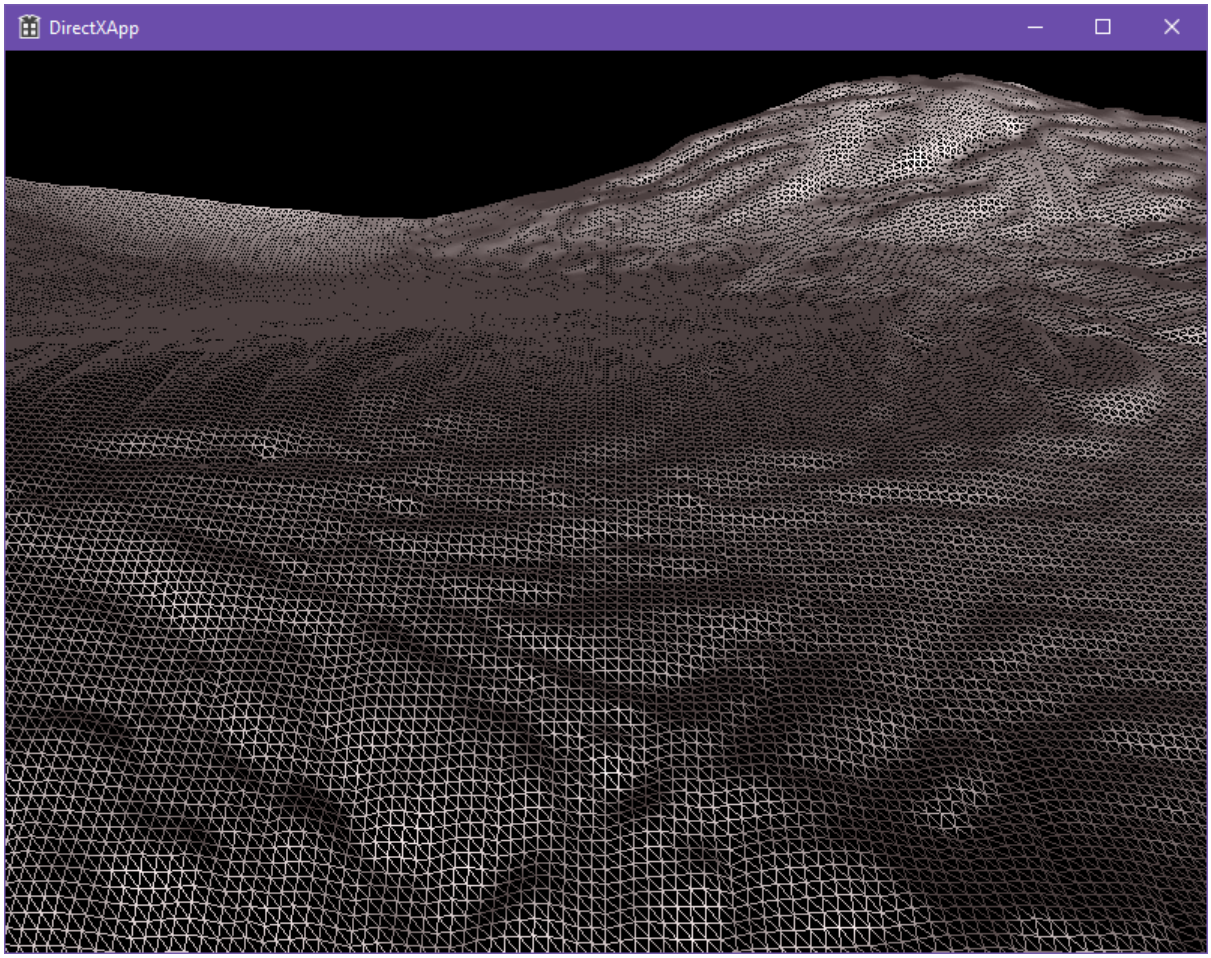
Because these are now normalised values, you should multiply the value in the height map by a suitable maximum world height value (I have used the `height` parameter with a value of 80 as a starting point) to give a suitable Y value. Note that the height map gives the height value for each vertex of the squares in the grid, so you need to be careful when determining which Y value to use for each vertex.

Since height values are saved in a list structure you need to compute the index `inx` in the list for a specific vertex `(i, j)` of the grid

```
unsigned int inx = j * _xnumVert + i; // index to height values list
```

The output of the executable

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:

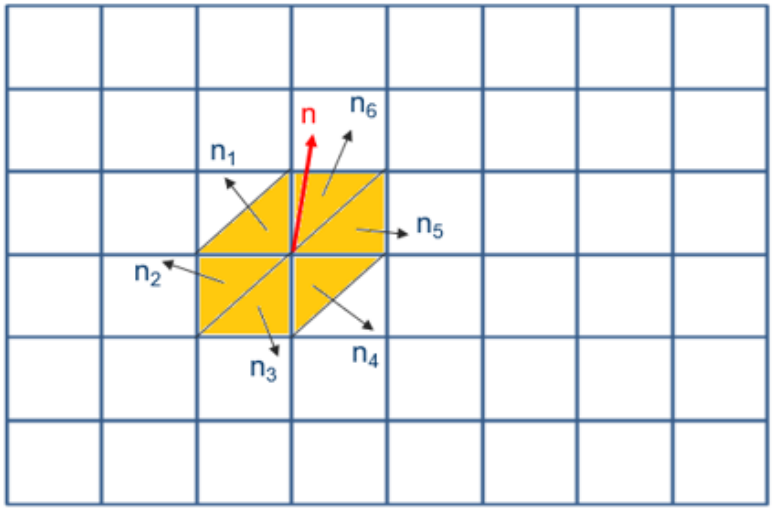


Step 4 – Calculating the Normals

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 upwards to (0,1,0) 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 triangle normal for each of the triangles that make up the square by using an outer vector product. Be careful to ensure that you calculate the two triangle 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 average of the triangle normals to the vertex normal for each of the vertices that make up the vertices of the terrain. In the diagram below, the arrows show the triangle normals which contribute to the normal of the vertex centred at the yellow polygon surrounding the vertex on which have to be added to.



$$\mathbf{n} = \frac{\sum_{i=1}^{k} \mathbf{n}_i}{k}$$

$$\hat{\mathbf{n}} = \frac{\mathbf{n}}{|\mathbf{n}|}$$

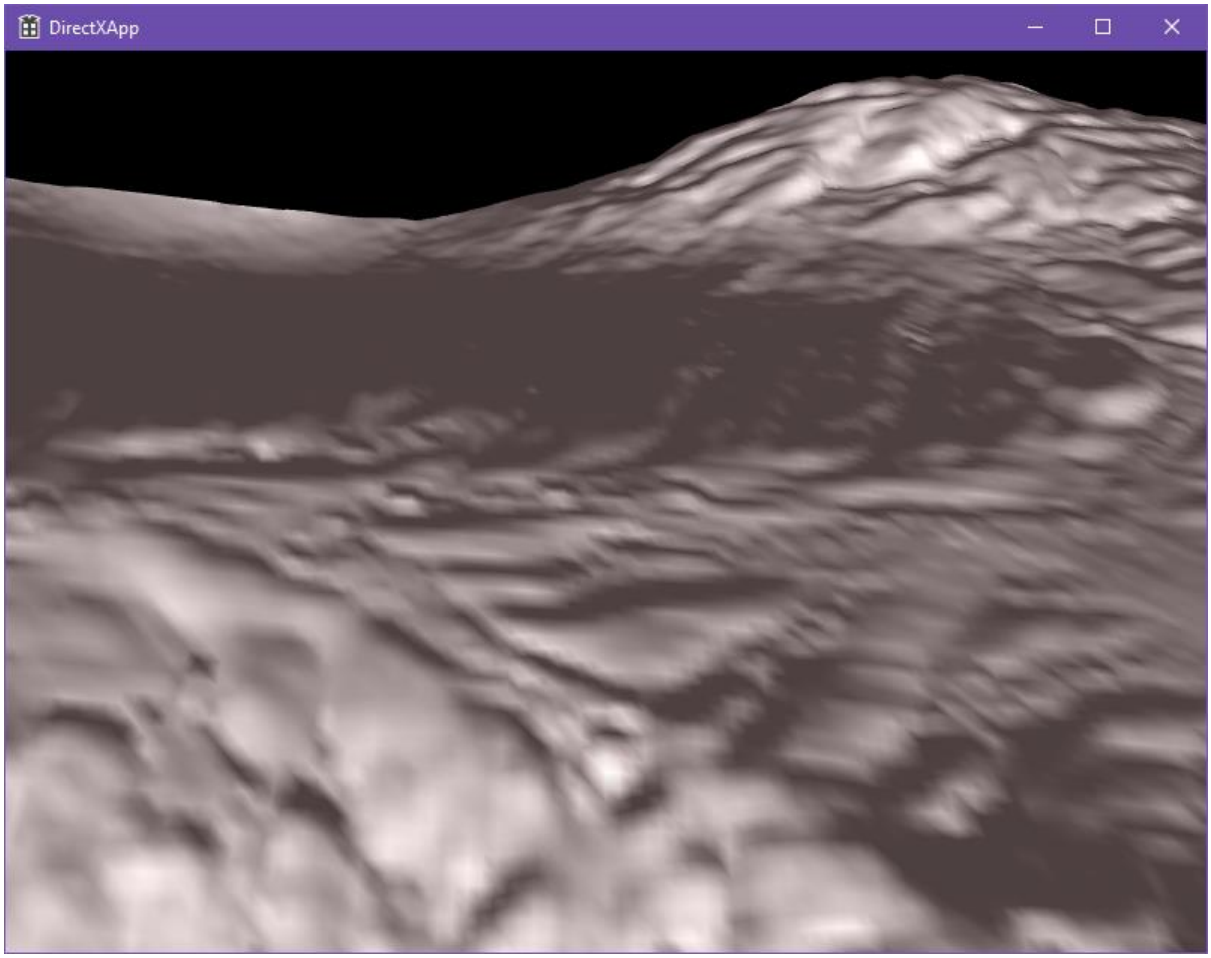
Once you have done this for every vertex in the grid, you should then go iterate through all of the vertices again and normalise the normal vectors.

```
// Calculate vertex normals
unsigned int* vertexContributingCount = new unsigned int[_numVert];
Vector3 u, v, normal;

for (unsigned int  index = 0; index < _numVert; index++)
{
	vertexContributingCount[index] = 0;
}
for (unsigned int  j = 0; j < (_znumVert - 1); j++)
	for (unsigned int  i = 0; i < (_xnumVert - 1); i++)
	{
		unsigned int index = (j * (_xnumVert - 1) + i) * 3 * 2;

		unsigned int index0 = terrainInds[index];
		unsigned int index1 = terrainInds[index + 1];
		unsigned int index2 = terrainInds[index + 2];
		u = XMVectorSet(terrainVerts[index1].Pos.x - terrainVerts[index0].Pos.x,
			terrainVerts[index1].Pos.y - terrainVerts[index0].Pos.y,
```

```
                terrainVerts[index1].Pos.z - terrainVerts[index0].Pos.z,
                1.0f);
            v = XMVectorSet(terrainVerts[index2].Pos.x - terrainVerts[index0].Pos.x,
                terrainVerts[index2].Pos.y - terrainVerts[index0].Pos.y,
                terrainVerts[index2].Pos.z - terrainVerts[index0].Pos.z,
                1.0f);
            normal = XMVector3Cross(u, v); //CCW
            //normal = XMVector3Cross(v, u); //CW
            XMStoreFloat3(&terrainVerts[index0].Norm,
            XMVectorAdd(XMLoadFloat3(&terrainVerts[index0].Norm), normal));
            vertexContributingCount[index0]++;
            XMStoreFloat3(&terrainVerts[index1].Norm,
            XMVectorAdd(XMLoadFloat3(&terrainVerts[index1].Norm), normal));
            vertexContributingCount[index1]++;
            XMStoreFloat3(&terrainVerts[index2].Norm,
            XMVectorAdd(XMLoadFloat3(&terrainVerts[index2].Norm), normal));
            vertexContributingCount[index2]++;

            unsigned int index3 = terrainInds[index + 3];
            unsigned int index4 = terrainInds[index + 4];
            unsigned int index5 = terrainInds[index + 5];
            u = XMVectorSet(terrainVerts[index4].Pos.x - terrainVerts[index3].Pos.x,
                terrainVerts[index4].Pos.y - terrainVerts[index3].Pos.y,
                terrainVerts[index4].Pos.z - terrainVerts[index3].Pos.z,
                1.0f);
            v = XMVectorSet(terrainVerts[index5].Pos.x - terrainVerts[index3].Pos.x,
                terrainVerts[index5].Pos.y - terrainVerts[index3].Pos.y,
                terrainVerts[index5].Pos.z - terrainVerts[index3].Pos.z,
                1.0f);
            normal = XMVector3Cross(u, v); //CCW
            //normal = XMVector3Cross(v, u); //CW
            XMStoreFloat3(&terrainVerts[index3].Norm,
            XMVectorAdd(XMLoadFloat3(&terrainVerts[index3].Norm), normal));
            vertexContributingCount[index3]++;
            XMStoreFloat3(&terrainVerts[index4].Norm,
            XMVectorAdd(XMLoadFloat3(&terrainVerts[index4].Norm), normal));
            vertexContributingCount[index4]++;
            XMStoreFloat3(&terrainVerts[index5].Norm,
            XMVectorAdd(XMLoadFloat3(&terrainVerts[index5].Norm), normal));
            vertexContributingCount[index5]++;
        }

// Now divide the vertex normals by the contributing counts and normalise
for (unsigned int  index = 0; index < _numVert; index++)
{
        Vector3 vertexNormal = XMLoadFloat3(&terrainVerts[index].Norm);
        if (vertexContributingCount[index] == 0)
            XMStoreFloat3(&terrainVerts[index].Norm, XMVectorSet(0.0f, 1.0f, 0.0f,
            1.0f));
        else XMStoreFloat3(&terrainVerts[index].Norm, vertexNormal /
        (float)vertexContributingCount[index]);
}
```

The output of the executable

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



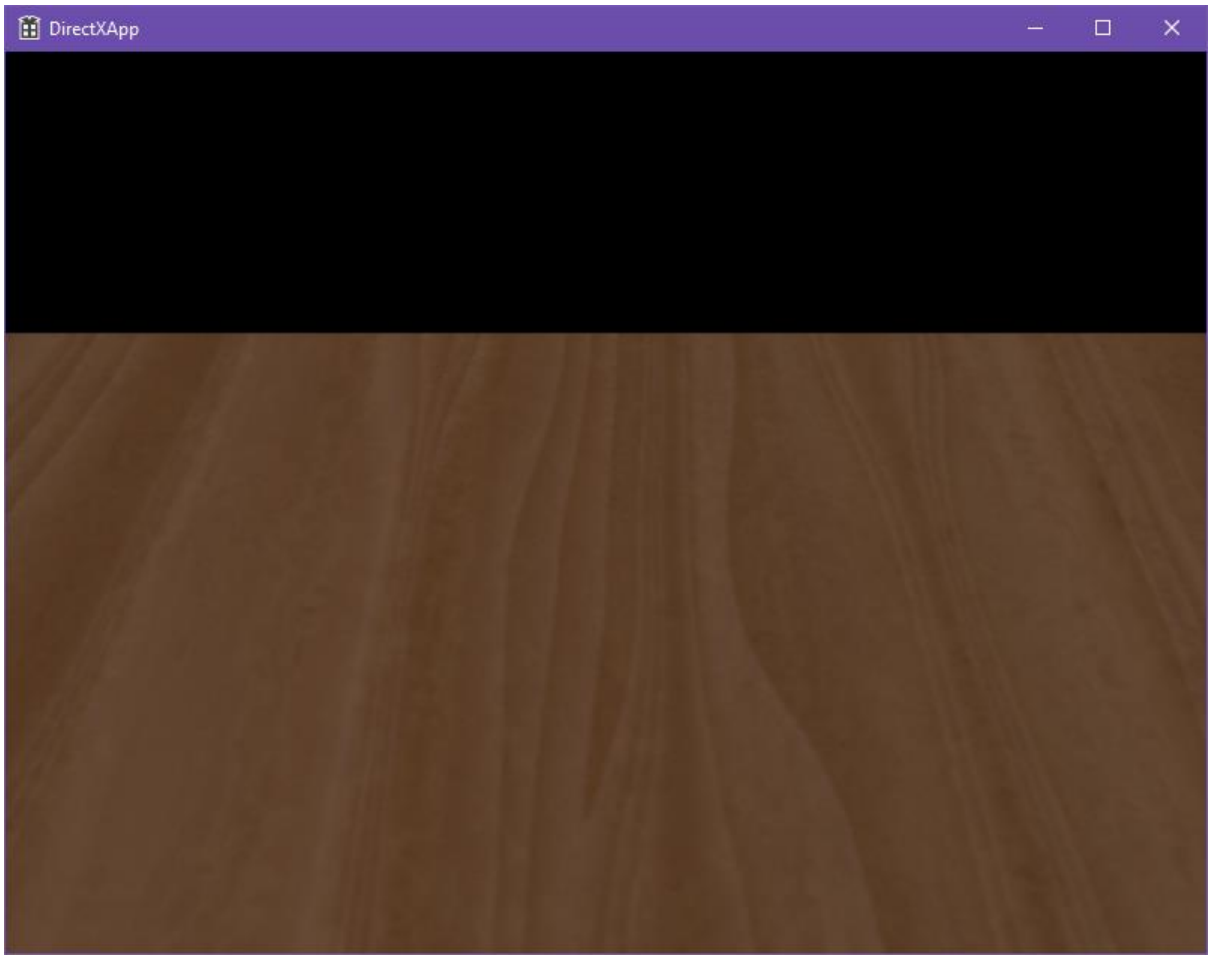
Step 5 – Applying different Textures

If you want to try applying a texture, we have provided some example textures that you can use. 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.

The output of the executable

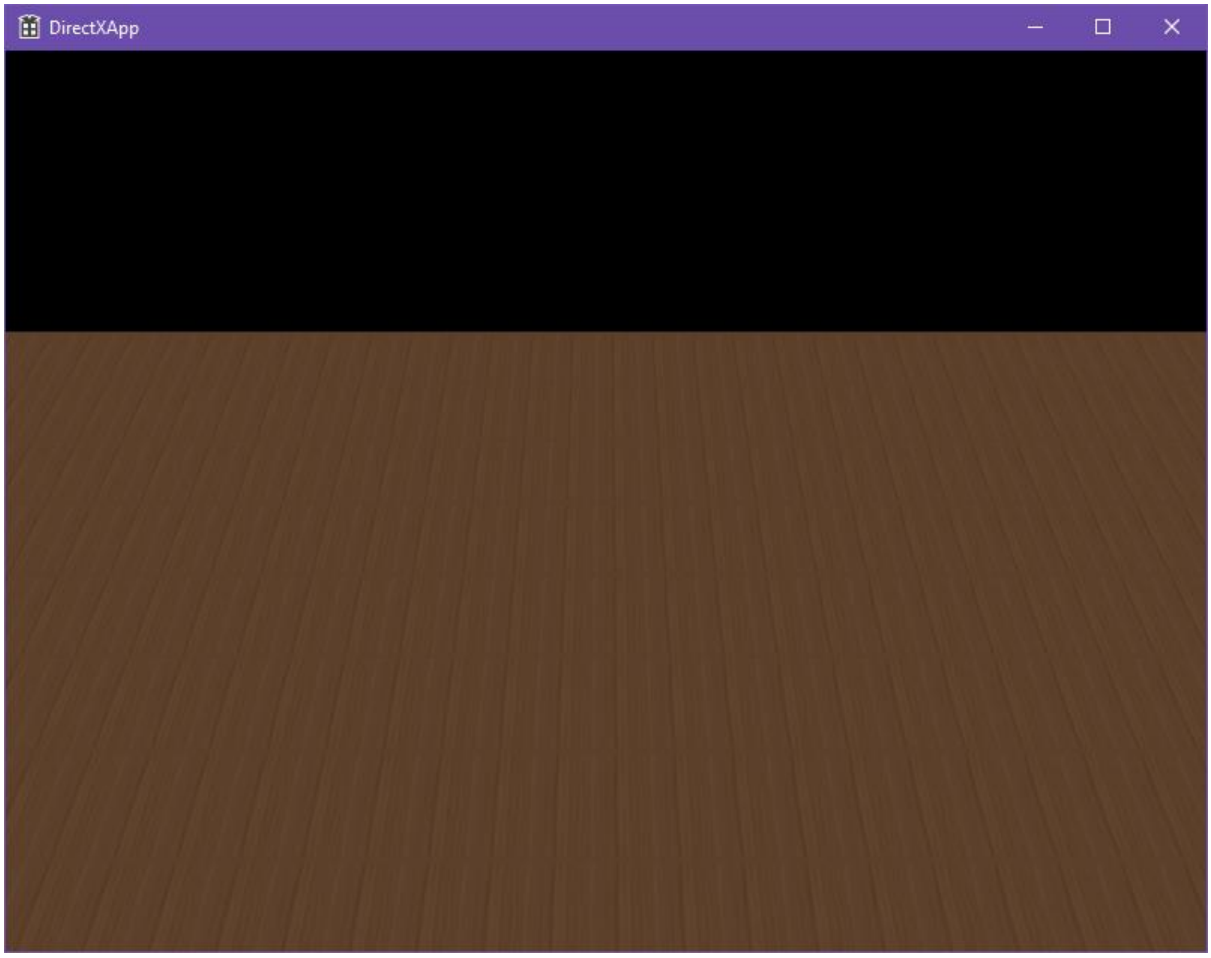
After compiling and running the executable, the application window should look like this:

Use of "Wood.png" texture stretched to the whole terrain



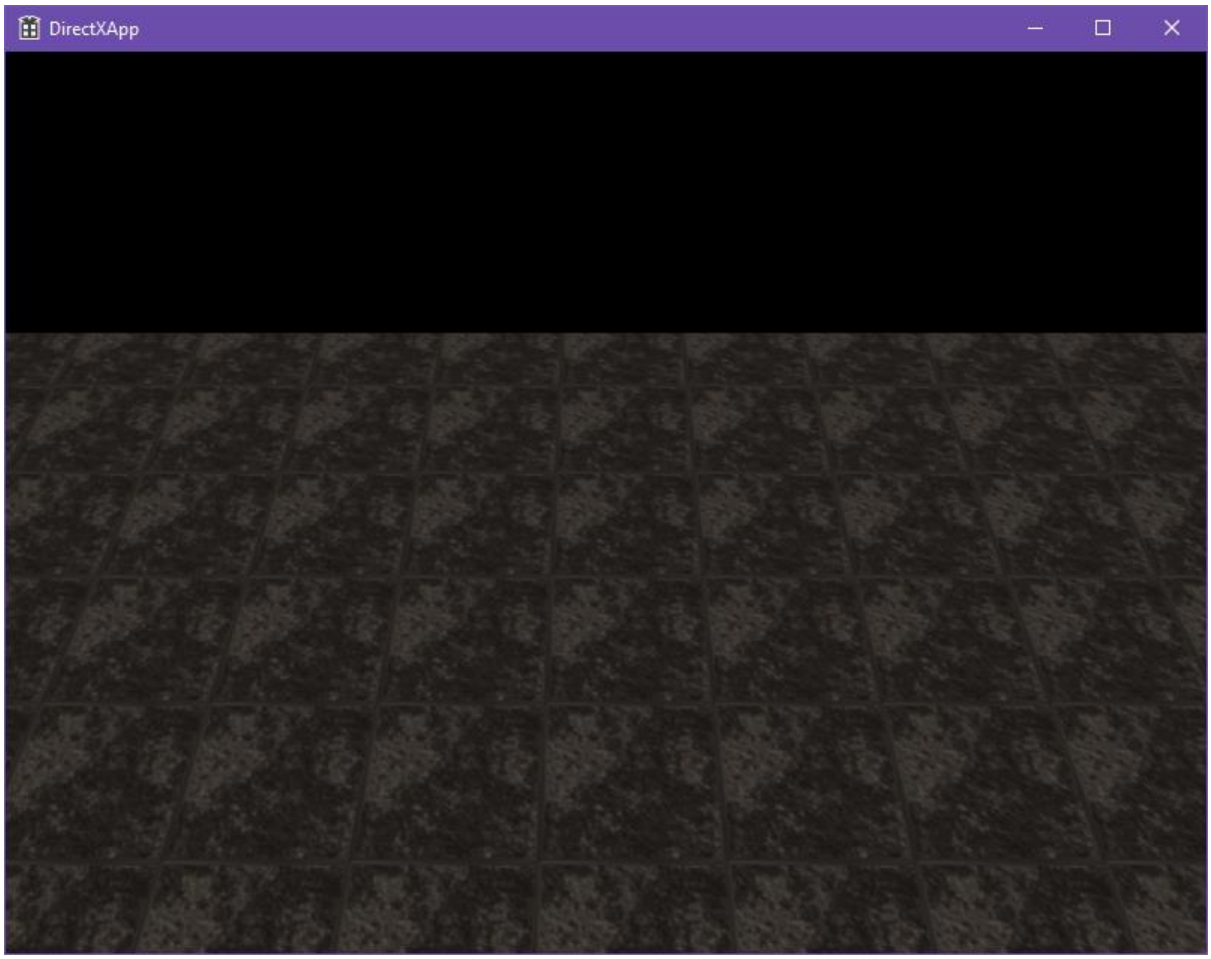
- ⇨ `BuildTexture(L"Wood.png");`
- ⇨ `_useHeightMap = false;`
- ⇨
```
//Stretched Texture
terrainVerts[inx].Tex.x = i / (float)_xnumVert;
terrainVerts[inx].Tex.y = j / (float)_znumVert;
```

Use of "Wood.png" texture tiling 1:2 to the whole terrain



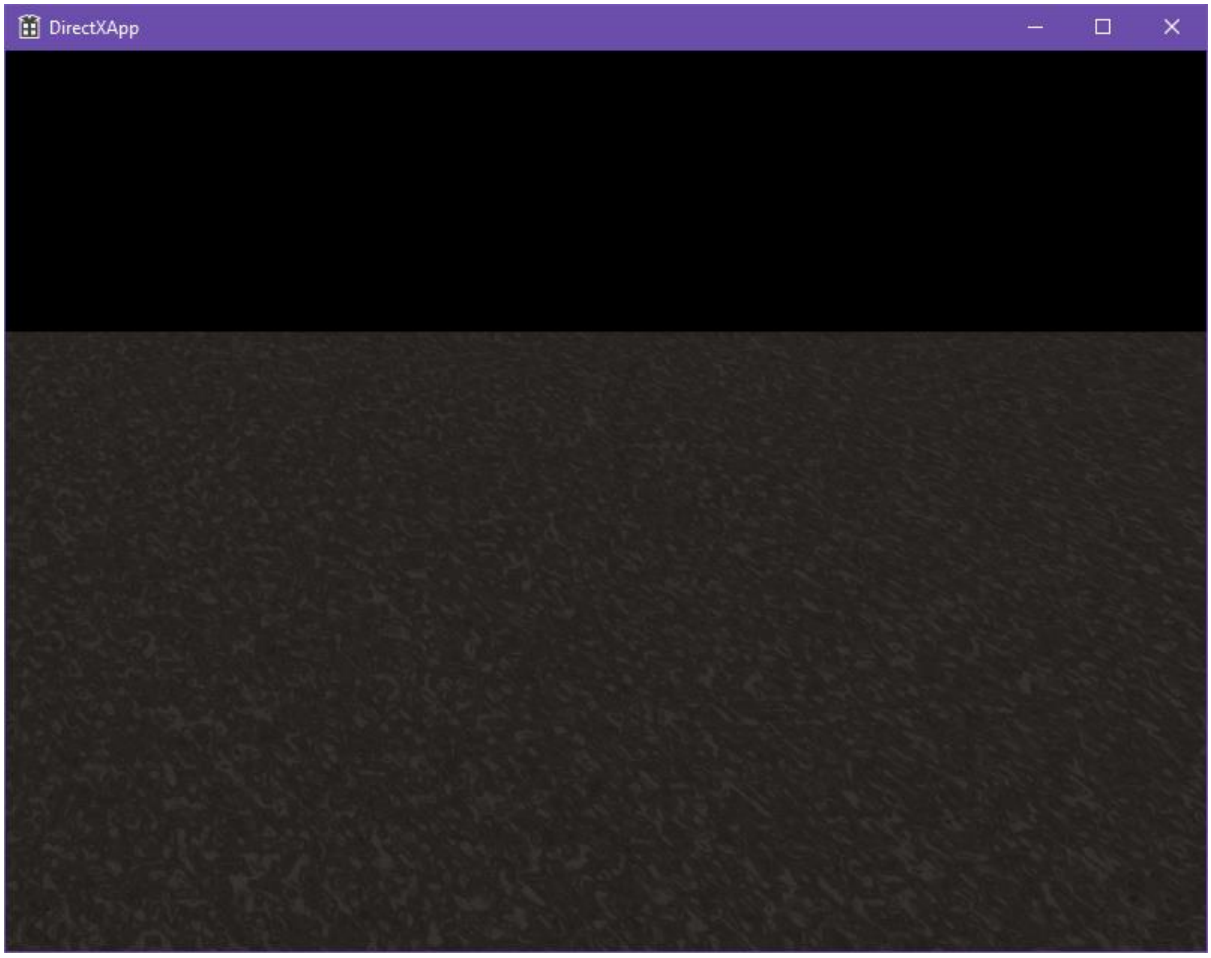
- ⇨ `BuildTexture(L"Wood.png");`
- ⇨ `_useHeightMap = false;`
- ⇨
```
//Tiling Texture
terrainVerts[inx].Tex.x = (float)(i % 8) / 8.0f;
terrainVerts[inx].Tex.y = (float)(j % 16) / 16.0f;
```

Use of "Concrete.png" texture tiling 1:1 to the whole terrain



- ⇨ `BuildTexture(L "Concrete.png");`
- ⇨ `_useHeightMap = false;`
- ⇨
```
//Tiling Texture
terrainVerts[inx].Tex.x = (float)(i % 24) / 24.0f;
terrainVerts[inx].Tex.y = (float)(j % 24) / 24.0f;
```

Use of "Concrete.png" texture with randomized sampling to the whole terrain

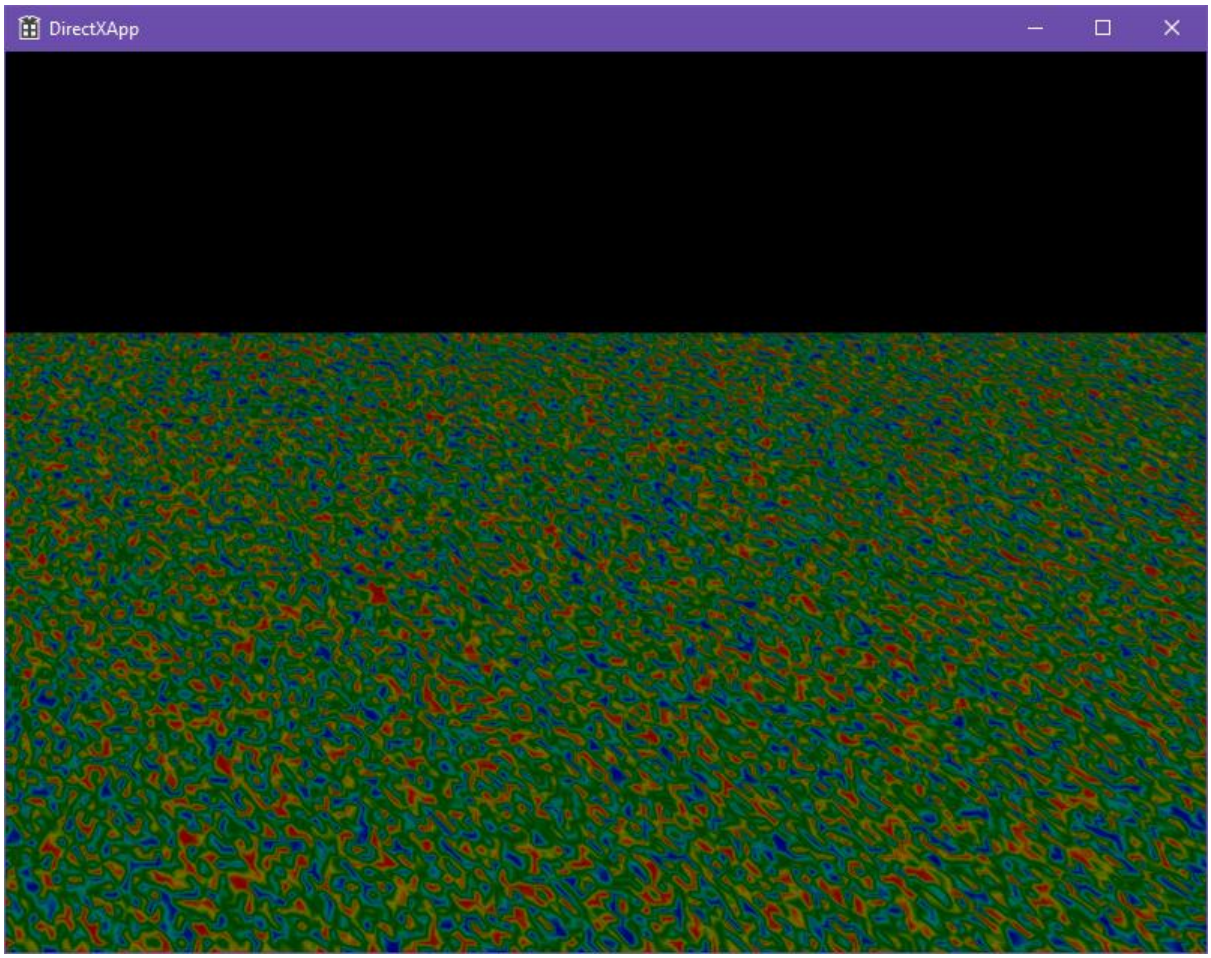


⇨ `BuildTexture(L "Concrete.png");`

⇨ `_useHeightMap = false;`

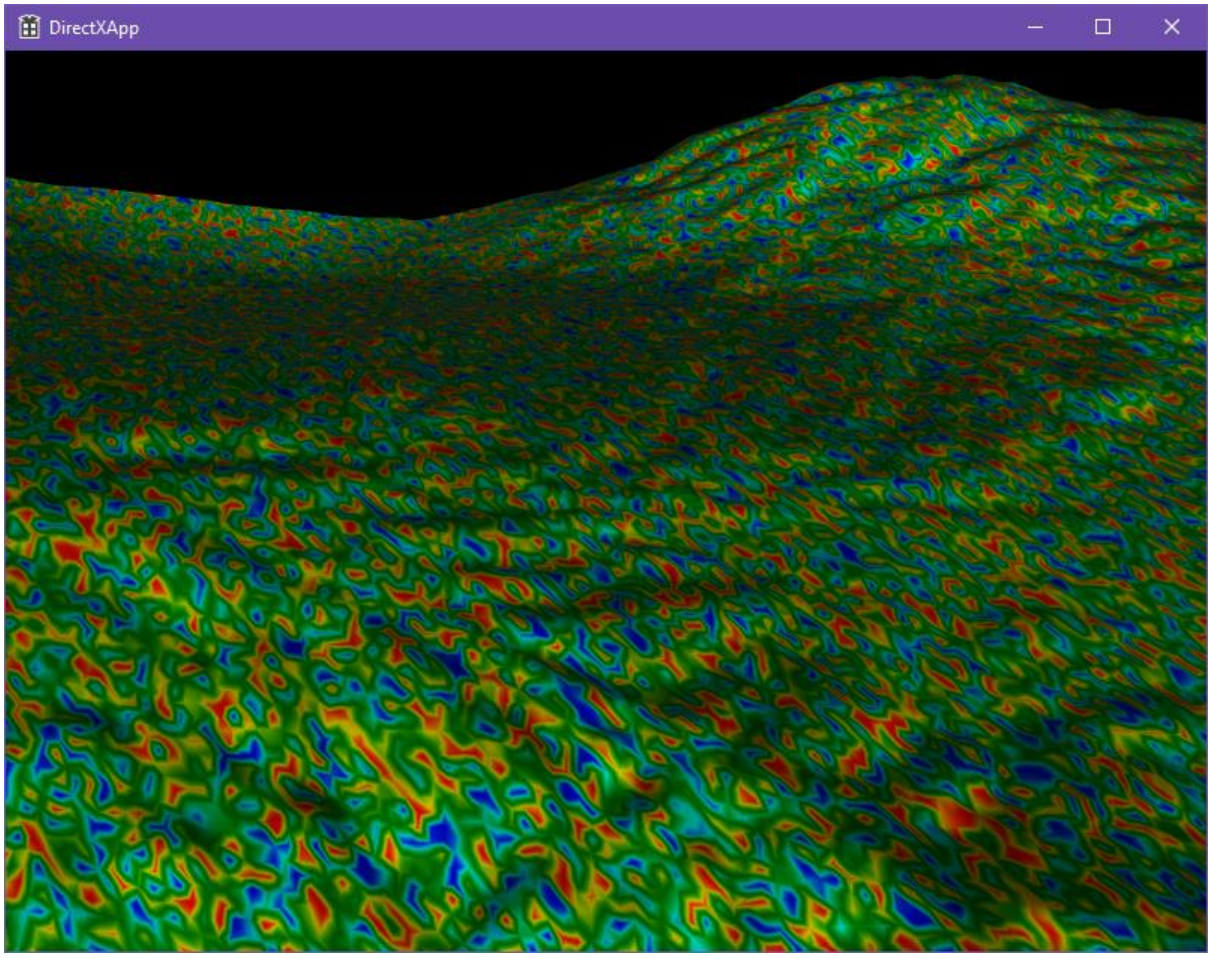
⇨
```
//Random Sampling of Texture
terrainVerts[inx].Tex.x = (rand() % 1000) / 1000.0f;
terrainVerts[inx].Tex.y = (rand() % 1000) / 1000.0f;
```

Use of "RGB_Colormap.png" texture with randomized sampling to the whole terrain



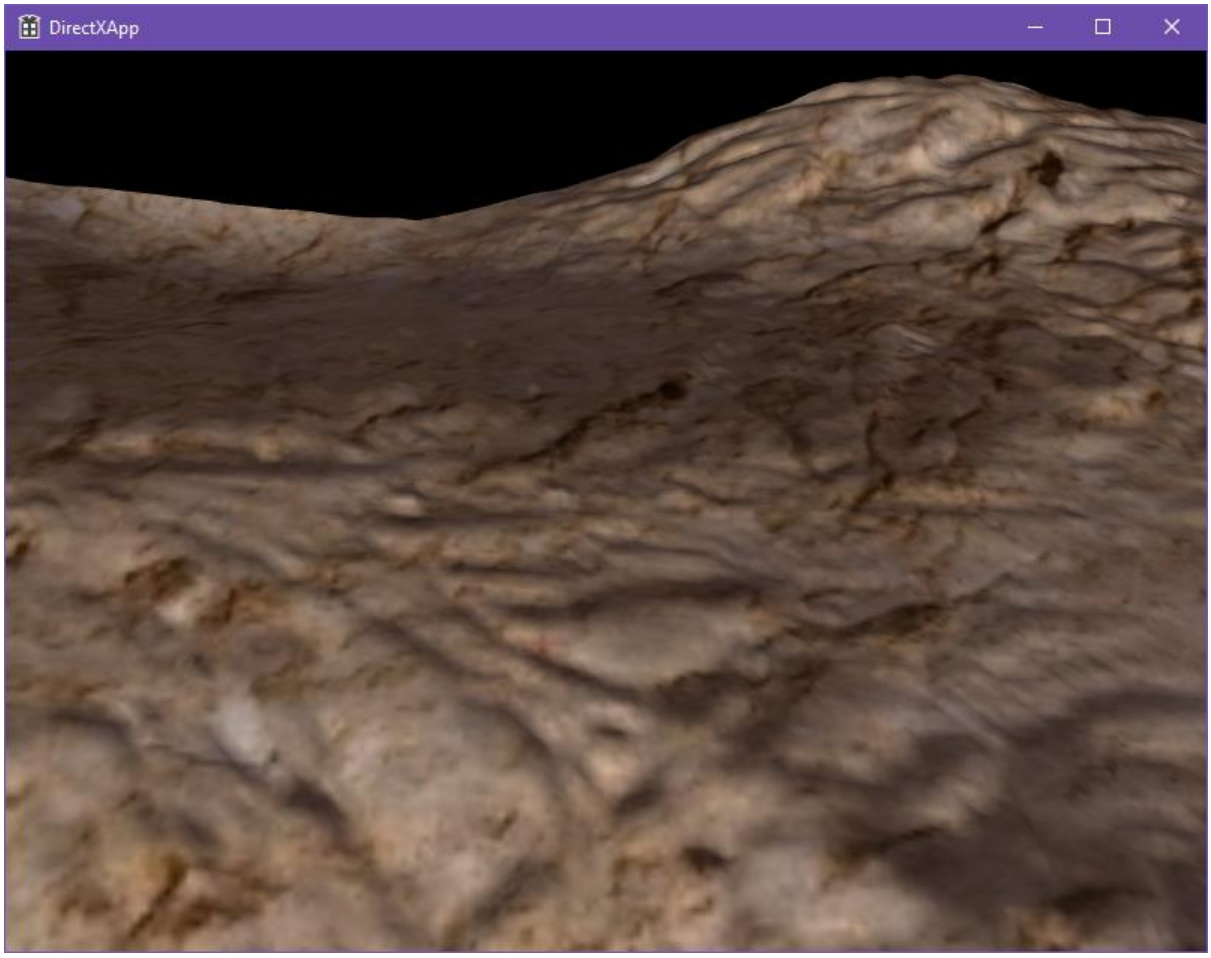
- `BuildTexture(L "RGB_Colormap.png");`
- `_useHeightMap = false;`
-
```
//Random Sampling of Texture
terrainVerts[inx].Tex.x = (rand() % 1000) / 1000.0f;
terrainVerts[inx].Tex.y = (rand() % 1000) / 1000.0f;
```

Use of "RGB_Colormap.png" texture with randomized sampling to the terrain with heightmap

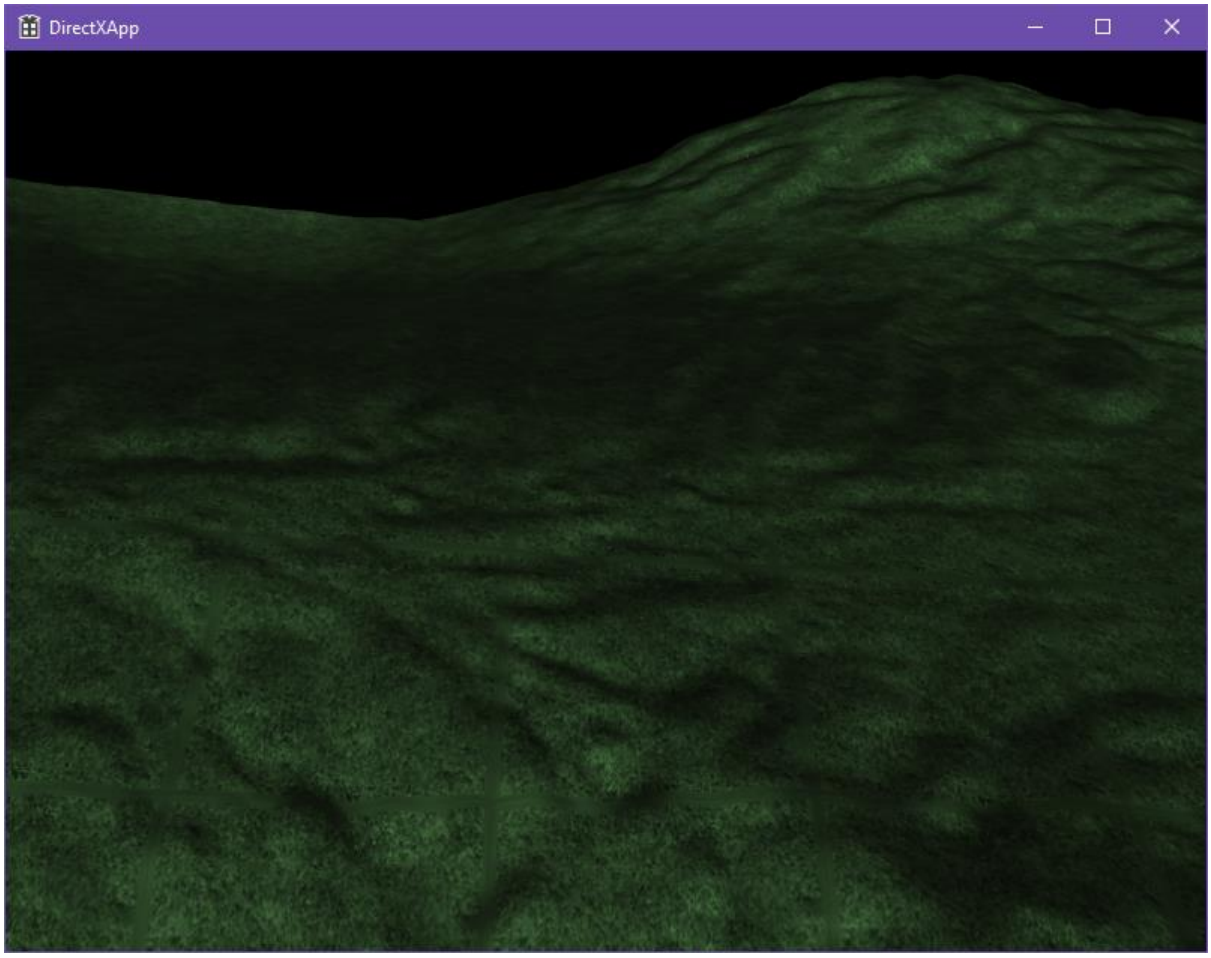


- ⇨ `BuildTexture(L "RGB_Colormap.png");`
- ⇨ `//_useHeightMap = false;`
- ⇨
```
//Random Sampling of Texture
terrainVerts[inx].Tex.x = (rand() % 1000) / 1000.0f;
terrainVerts[inx].Tex.y = (rand() % 1000) / 1000.0f;
```

Use of "dirt.jpg" texture stretched to the whole terrain with heightmap



Use of "grass.jpg" texture tiling 1:1 to the whole terrain with heightmap



Note: A fully working code for the implementation of a textured terrain with a height map is provided to you in the `Exercise_06_1.zip` *file.*