UNIVERSITY of DERBY

# Pixel Shaders and Blending Techniques

Wayne Rippin (UoD)
Dr Panagiotis Perakis (MC)

# This Week

- Moving to using calculating the pixel colour in the pixel shader
- Introduction to blending
- Terrain blending

- Note that this is a just a starting point.  We could take all of these much further, but that is beyond the scope of this module.

# Doing more in the Pixel Shader

- Until now, we have only been doing the colour calculations in the vertex shader.
- This gives us Gouraud shading
- However, this does not work well if we want to introduce specular highlighting
- For good specular highlighting, we need to move to doing our colour calculations in the pixel shader

# Doing more in the Pixel Shader

- To do the calculations in the pixel shader, we need to add the adjusted normal and world position of the object to the output from the vertex shader
- These will then be passed into the pixel shader and can be used for the lighting calculations
- The tutorial for this week takes you through the steps that are necessary.

# Blending

- There are times when you need to blend textures according to a specific algorithm
- One example of this is when we have to deal with transparent textures

# Example

# Example

- This is made up of meshes that are fully opaque (opacity = 1.0) and meshes that have transparency (opacity < 1.0f).


# Blending

- Normally, when we render an object, DirectX looks at the pixel that is about to be rendered to see if it is further away than the pixel that is already on the screen (i.e. it looks at the transformed Z position).
- If the pixel is further away, then it is not drawn so that we see the objects that are supposed to be in the foreground correctly.
- However, sometimes this is not exactly what we want.  Blending allows us to blend the source pixel (the one we are about to draw) with the destination pixel (the one already on the screen) using an equation.
- For example, when drawing a transparent object, we want some of what is already on the screen to show through even though the transparent object s in front of it.

# Blend States

- What we need to do is create a ‘blend state’ by populating a D3D11_BLEND_DESC structure and then calling CreateBlendState().
- We then tell DirectX to use this blend state when rendering by making a call to OMSetBlendState().
- Example.
- Note that we need to render the opaque meshes first and then render the transparent meshes.

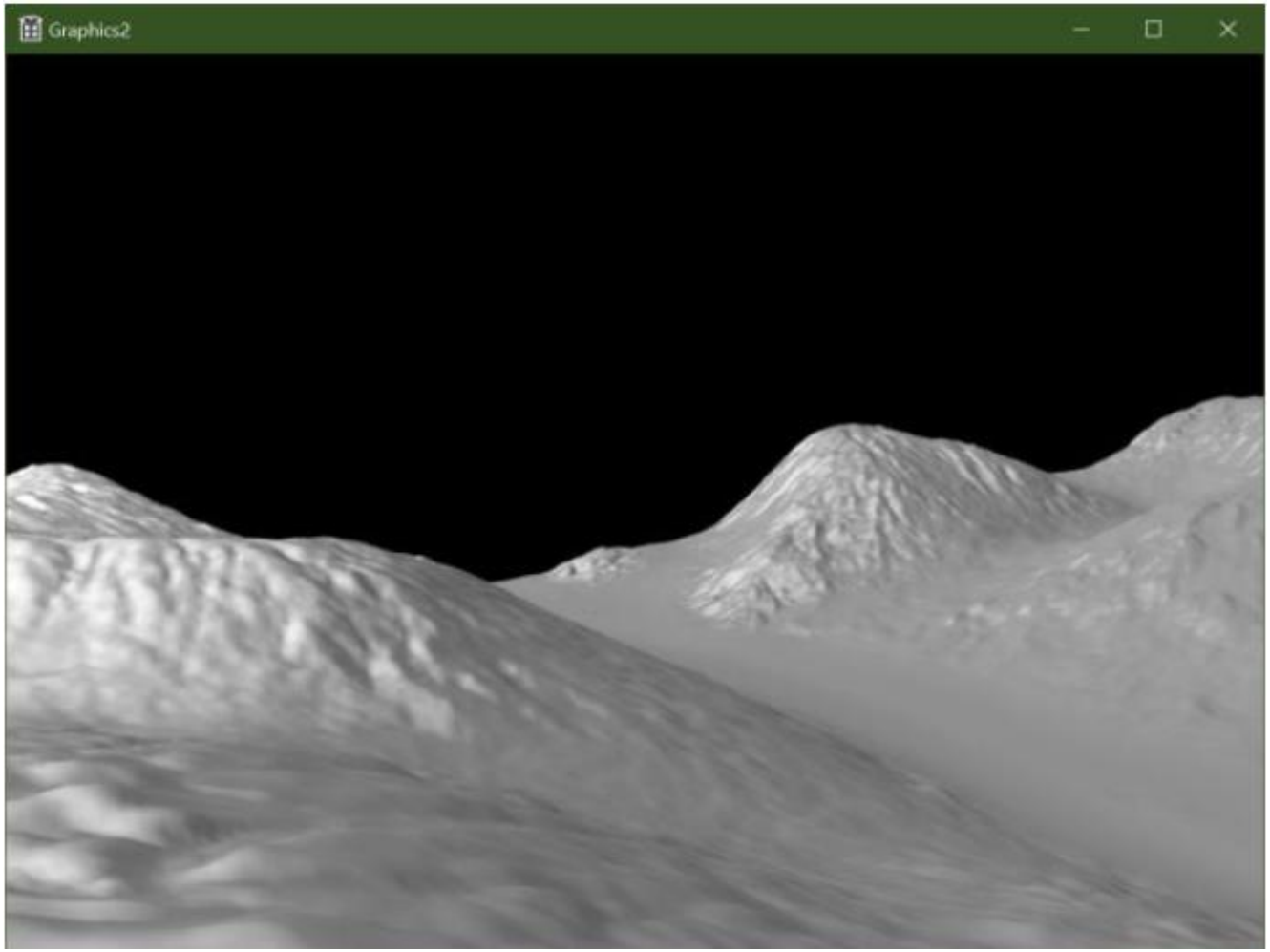
# Further Reading

- To cover blending in detail would take a few lectures all on its own, but here is some further reading:
  - https://www.braynzarsoft.net/viewtutorial/q16390-12-blending
  - https://learn.microsoft.com/en-us/windows/win32/api/d3d11/ne-d3d11-d3d11_blend
  - https://ycpcs.github.io/cs470-fall2014/labs/lab09.html

# Terrain - Where We Are

- We have already seen how the basic structure of a terrain mesh could be created.
  - A grid was built
  - A height map was applied (or you could use procedural terrain generation)
  - Normals were generated for the vertices in the grid

# Terrain - Where We Are

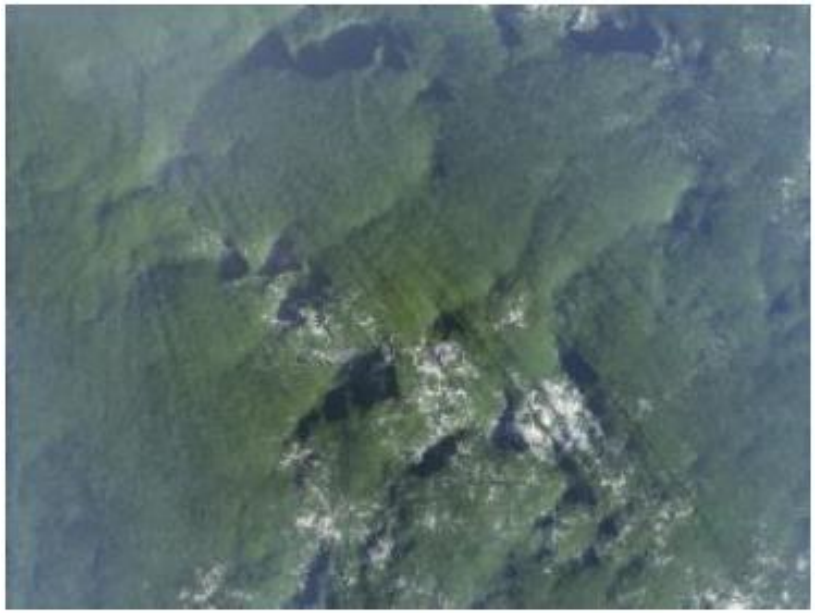


# Terrain - Where We Are

- Unless we are modelling the surface of the moon, this isn't really ideal
- What we need to do now is apply texture to the terrain

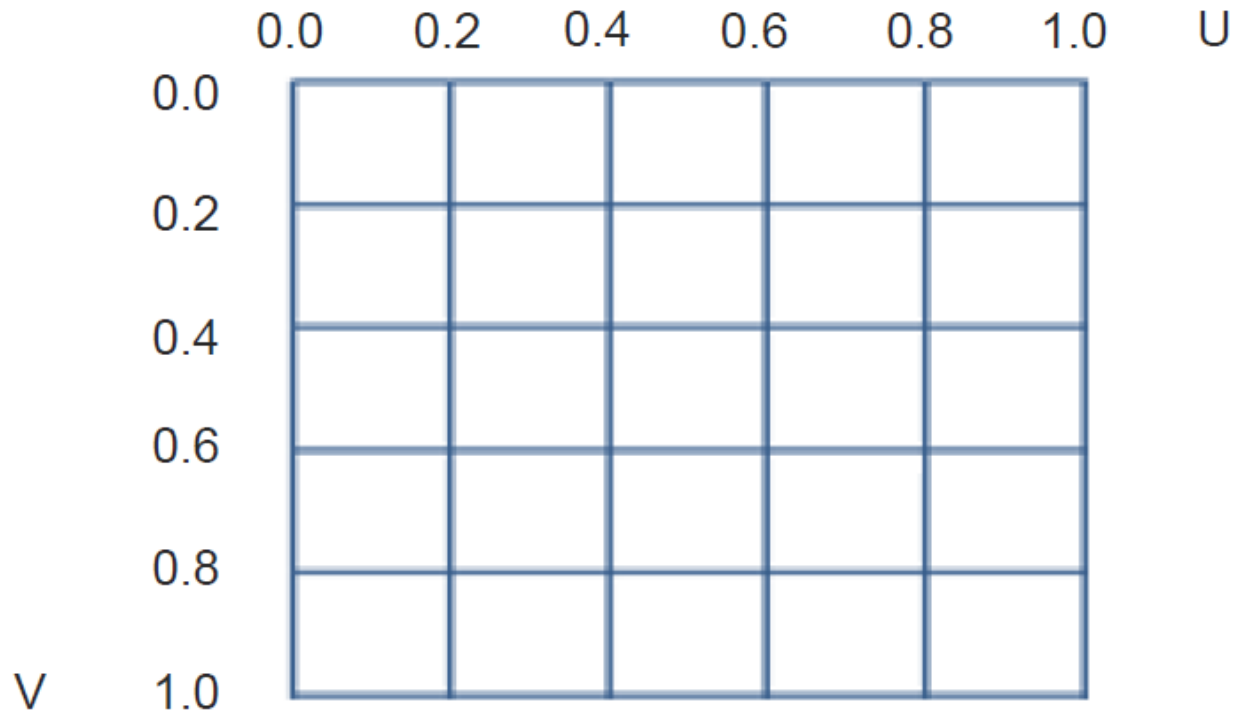
# Texturing Terrain - The Simple Approach

- Use a large texture map and stretch it over the entire grid
- Texture map can be produced by a terrain generator
- Increase detail and change image size to maximum possible
- Each vertex has UV values in the range 0.0 to 1.0

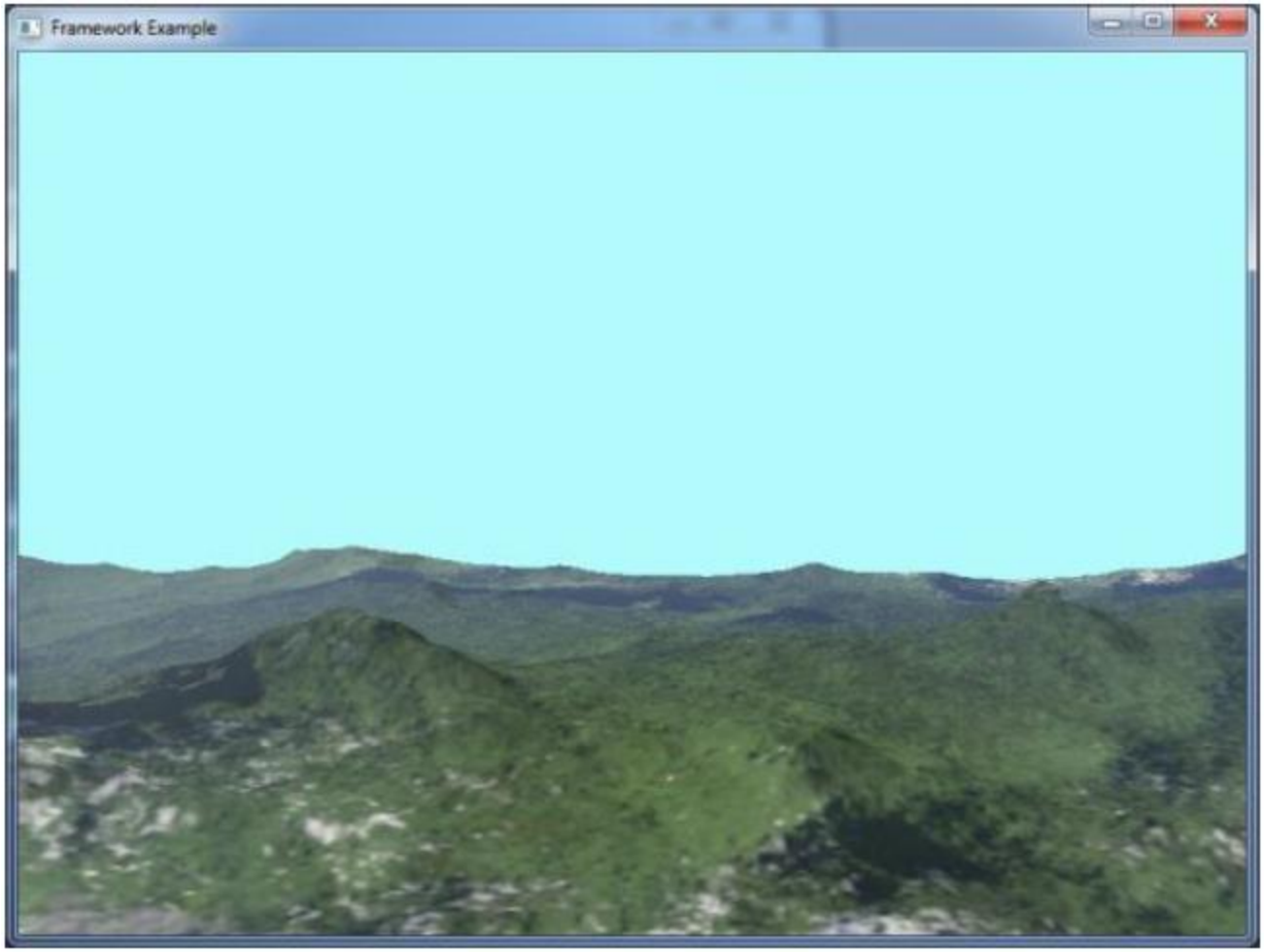


# Texturing Terrain - The Simple Approach

- So for a 5 x 5 grid, the UV coordinates would be:



# Texturing Terrain - The Simple Approach



# Texturing Terrain - The Simple Approach

- This can be done relatively easily, but it does have some problems
- The texture maps can be very large
- Even for very large maps, the terrain for a particular square in the grid will only consist of one or a few pixels. This means that close-ups of the terrain can be very blurred
- Does not work for procedurally generated terrain.

# Texturing Terrain Procedurally

- Another solution is to generate a blend map and apply multiple textures maps to each square in the grid.
- The blend map determines how much of each texture contributes to the final result
- Each vertex has to have two sets of texture coordinates:
  - One set gives the position in the blend map for that vertex. This set would be determined just like the way the UV coordinates are calculated for the simple approach
  - The other set gives the position into the textures that are applied to each square in the grid. The UV values at the four corners of each square are

    (0, 0), (0, 1), (1, 0) and (1, 1)

# The Vertex Structure

```
struct TerrainVertex
{
    Vector3 Position;
    Vector3 Normal;
    Vector2 TexCoord;
    Vector2 BlendMapTexCoord;
};
```

# The Corresponding Input Layout

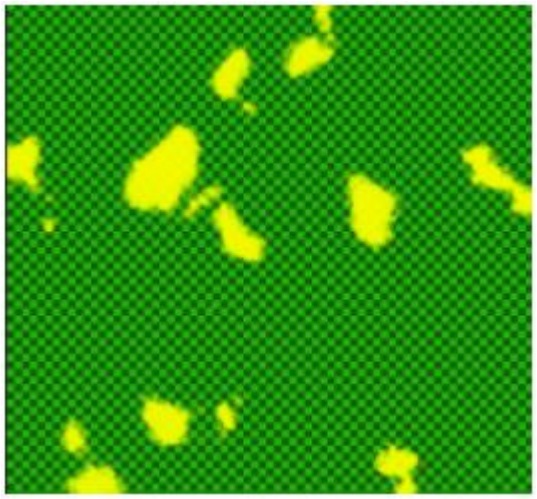
```
D3D11_INPUT_ELEMENT_DESCvertexDesc[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
      D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0,
      D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA,
      0 },
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0,
      D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA,
      0 },
    { "TEXCOORD", 1, DXGI_FORMAT_R32G32_FLOAT, 0,
      D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA,
      0 }
};
```

# The Input To The Vertex Shader

```
struct VertexShaderInput
{
    float3 Position : POSITION;
    float3 Normal : NORMAL;
    float2 TexCoord : TEXCOORD0;
    float2 BlendMapTexCoord : TEXCOORD1;
};
```

# The Blend Map

- We apply five textures to each square (in this case)
- The R, G, B and A components from the corresponding pixel in the blend map is used to determine how much of the top four textures contribute to the texture that is applied to the terrain
- A value of 0 indicates that the texture is not used at all.  A maximum value (255 in our case) means that the texture fully contributes.



# Using the Blend Map

A value from blend map determines how much this texture contributes

B value from blend map determines how much this texture contributes

G value from blend map determines how much this texture contributes

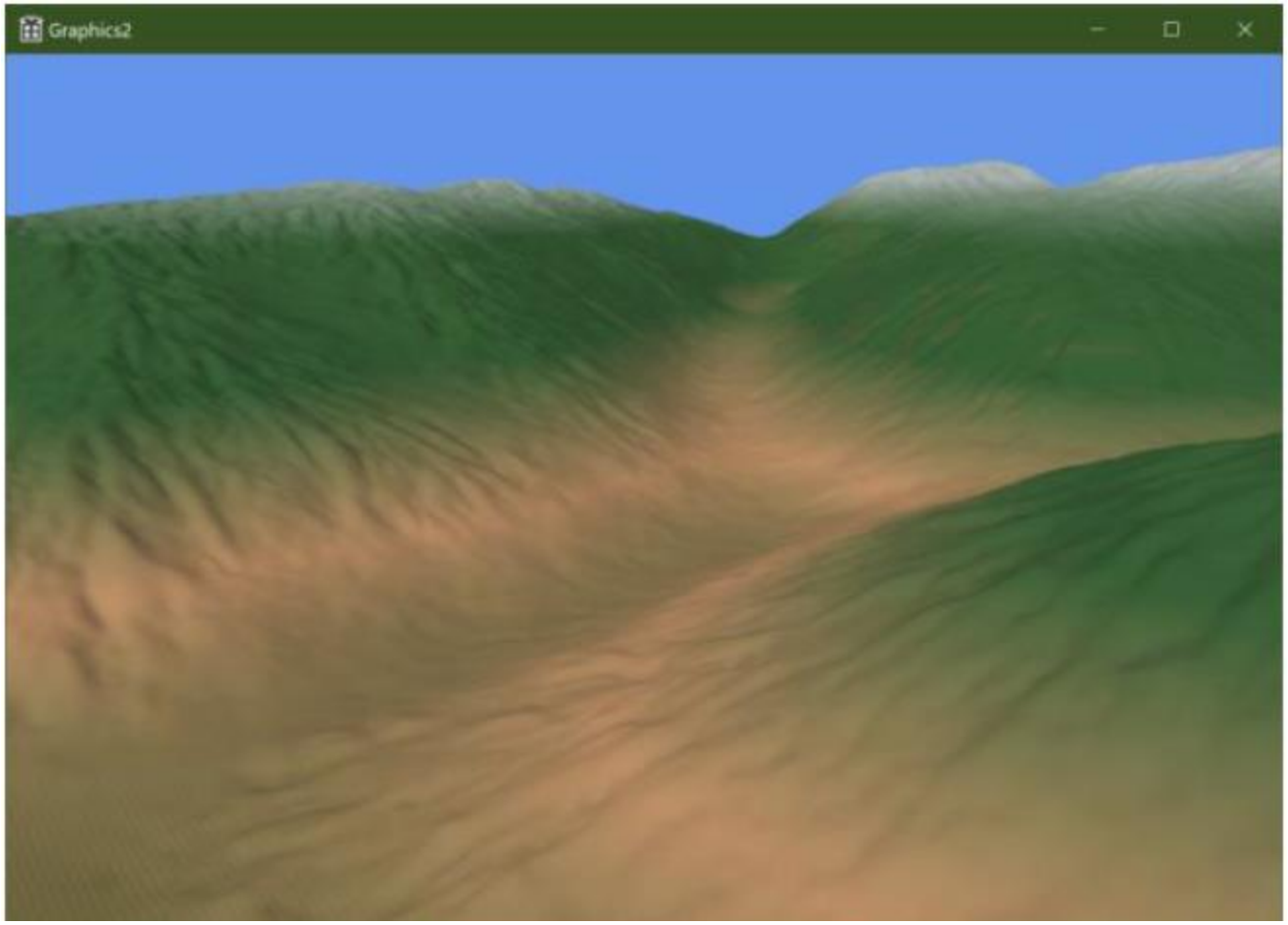
R value from blend map determines how much this texture contributes

Blending result

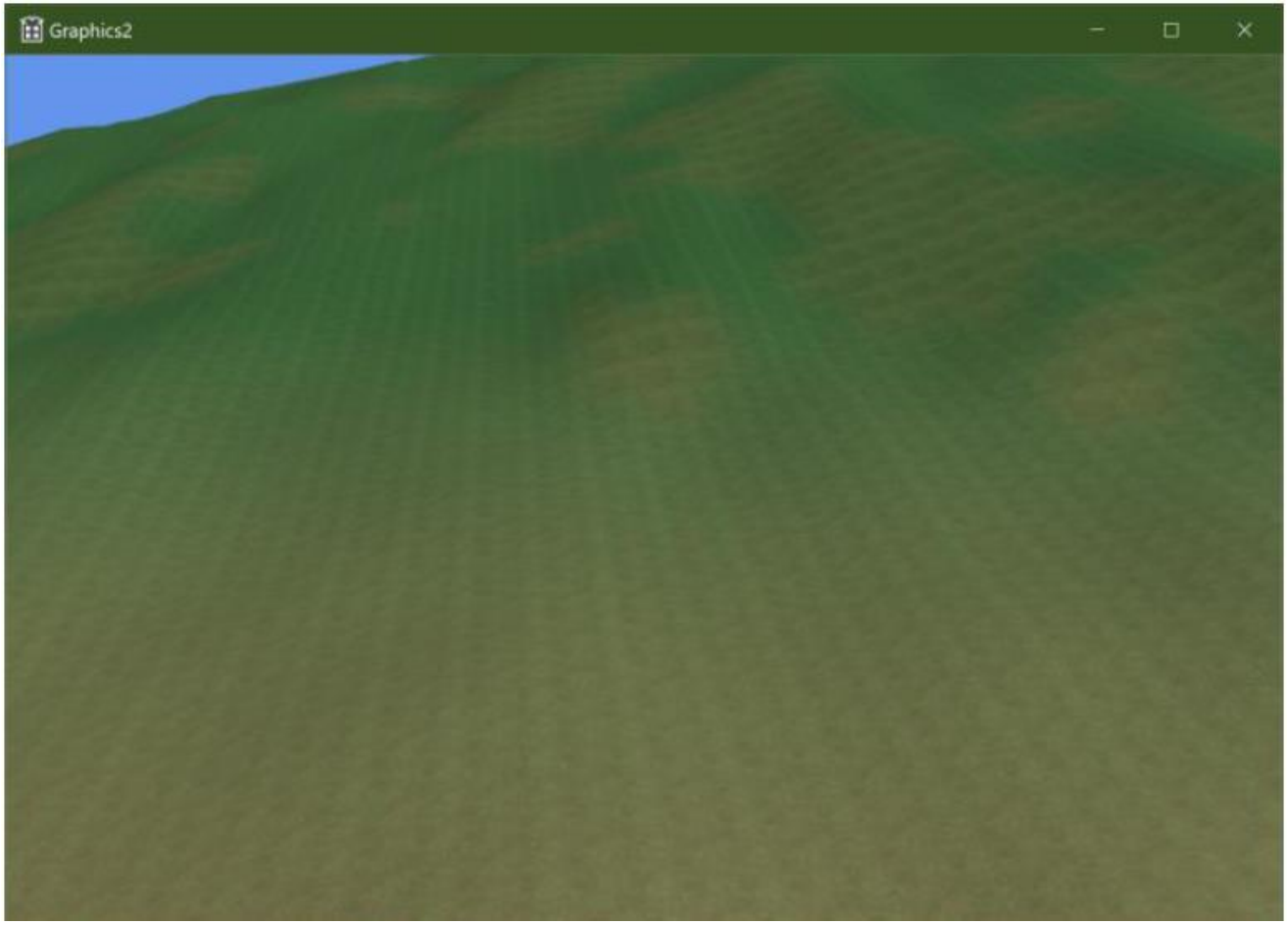
# Producing the Blend Map

- It can be produced using an art package, but this does not help for procedurally generated terrain
- We can also produce it programmatically, calculating the colour components at each pixel based on data in each vertex to determine the average height in each square, the slope, etc.
- To determine the amount of each texture based on the height, we need to calculate the average height for each square in the grid using the average of the Y values for each vertex that makes up the square and then use that value to determine how much of each texture we want to show
- To determine values based on slope, we use the Y value of the normal for the square. A value of 1.0 means that the normal is straight up –meaning that the square is flat. A value of 0.0 means that the normal is flat –meaning that the square is on its side. We use appropriate values between 0 and 1 depending on what value of the slop we are interested in.

# The Resulting Terrain



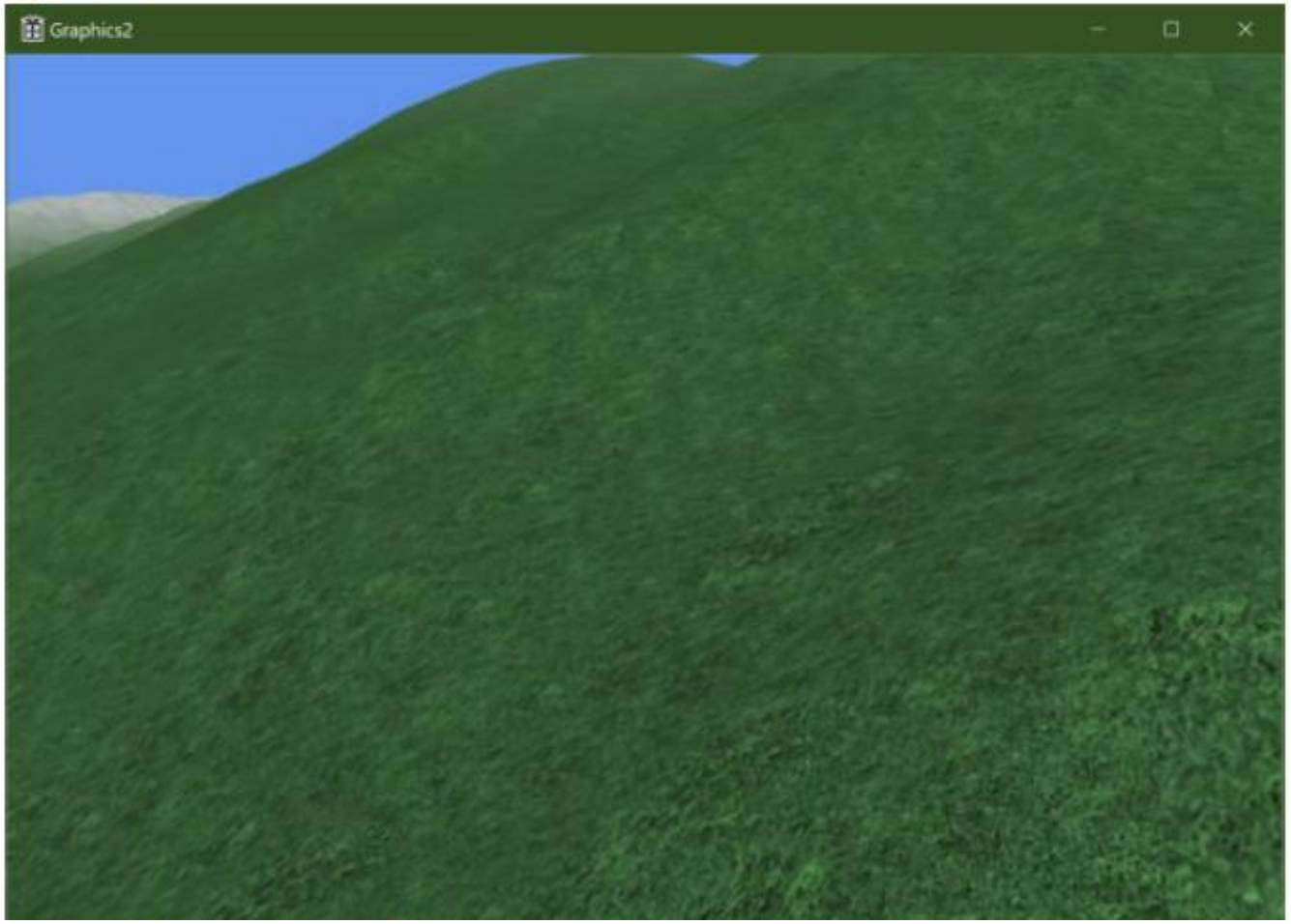
# One Problem



# The Problem

- Because the UVs in to the textures are all 0 or 1, depending on the textures you can end up with an unnatural looking grid pattern if you get too close.
- The solution to this is to change the UV values so that they are not 0 and 1 for each square, but are sensible random values between 0 and 1. This also has the impact of varying the texture even more.

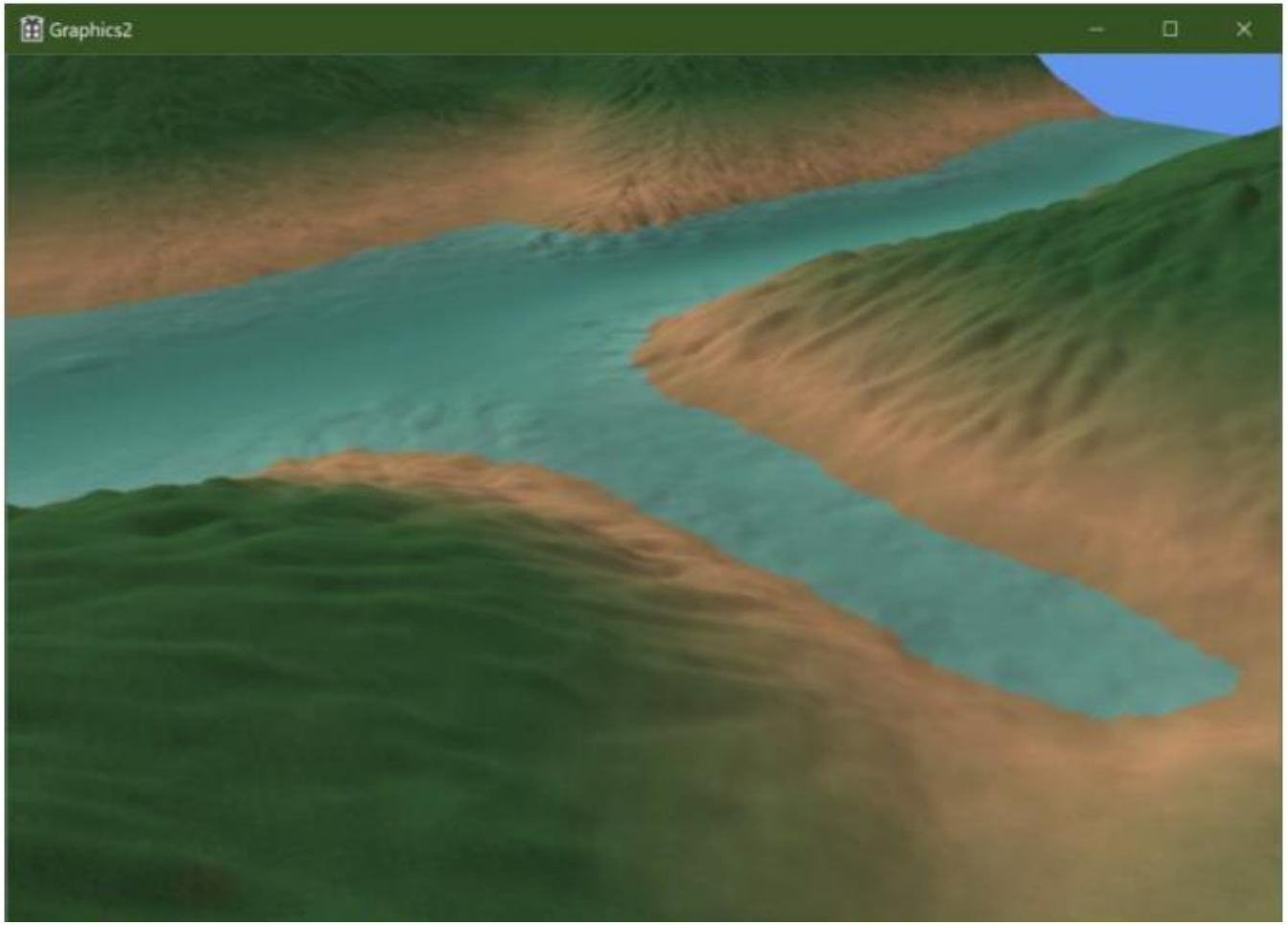
# The Result



# Adding Water

- If we want part of the scene to be under water, we need to determine what height we want to be the water level in the scene
- The simplest approach then is to adjust the vertex shader to set the height of any vertex that is under water to be the same height.
- We can also adjust the pixel shader to apply a tint to any terrain that is at that height.

# The First Attempt

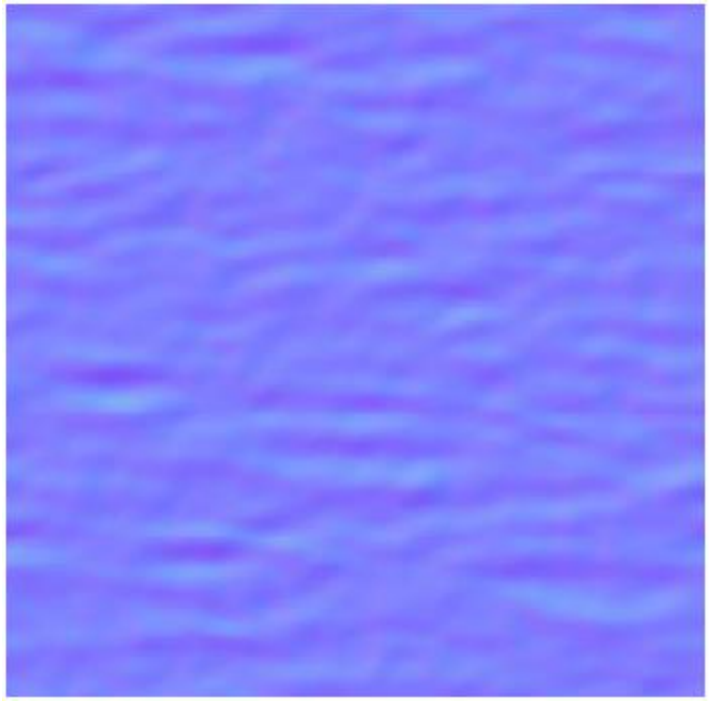


# The First Attempt

- This is a start, but it is not ideal.
- The biggest issue is that the lighting information is still coming from the normal for the terrain that is under the water and not the water itself.
- We really want to simulate the impact of light reflecting off ripples on the water.
- To do this, we use a *normal map*.

# Normal Maps

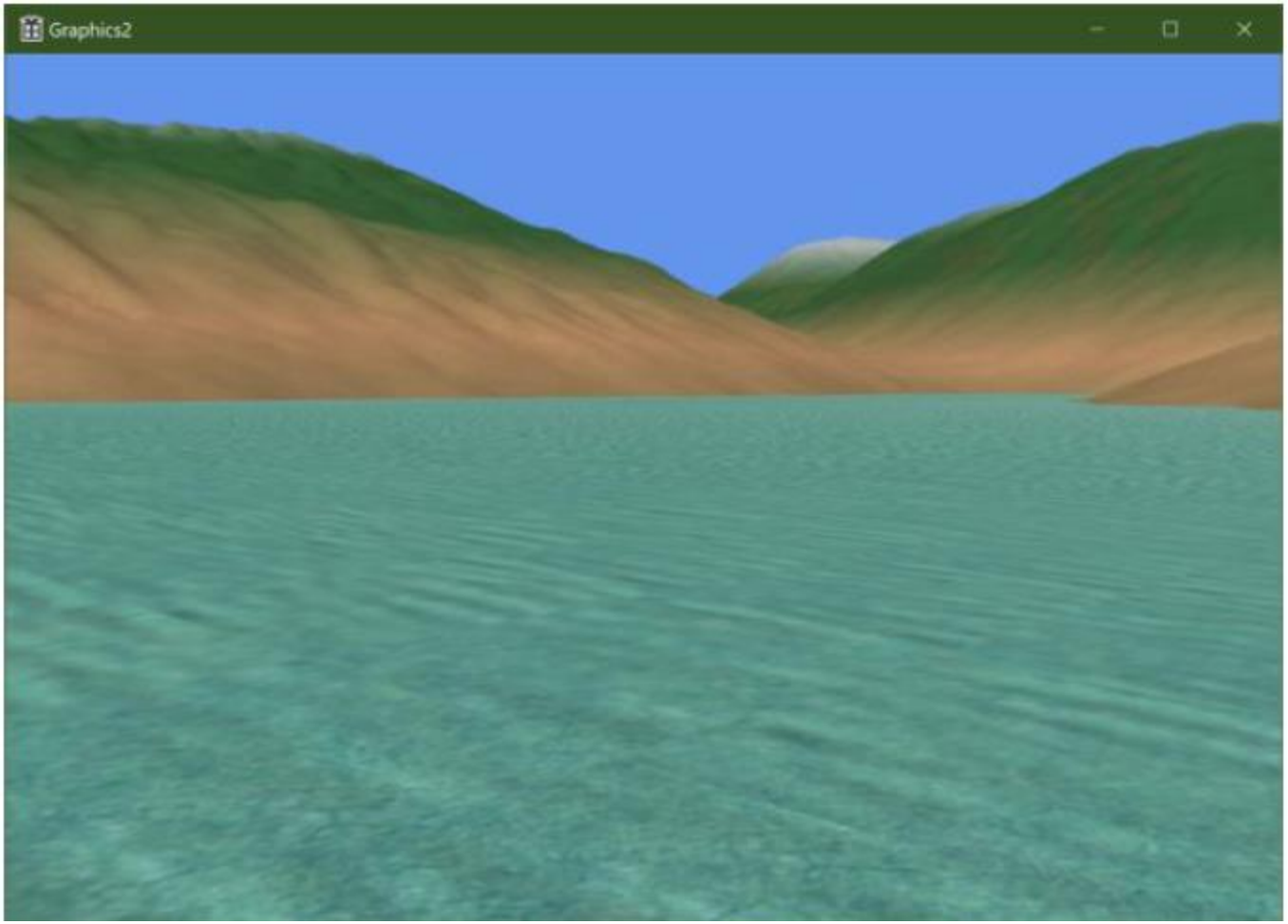
- A normal map is a texture where the RGB values do not represent the colours at that location.  Instead, the RGB value represents the X, Y and Z values of the normal at that location.



# Applying a Normal Map

- We load the normal map just like any other texture and supply it to the pixel shader
- If we are in a part of the terrain that is under water, for our initial attempt, we can extract the normal for the location from the normal map using the same UV values that we used for the textures
- We also increase the shininess value for this location.

# The End Result



# This is only a Start

- This is only a start on representing water
- We really need to deal with the refraction effect of water on things under the surface
- We also need to deal with reflection of surrounding terrain and sky on the water
- Much of this is beyond the scope of this module, but we will look at how we represent the sky next week.

# This Week

- The tutorial continues on how to apply texture to your terrain.